AllJoyn™ System Description

Version 14.06

September 26, 2014

Contents

Figures

Tables

1 Introduction

1.1 Document scope

This document provides system level description for AllJoyn™ system.

1.2 Document organization

This document is organized in following chapters:

Chapter 1 – Introduction: Provides introductory information including scope, document organization, definitions and acronyms, references, and document revision history.

Chapter 2 – System Overview: Gives an overview of the AllJoyn system along with key concepts.

Chapter 3 – System Architecture: Provides the system topology, the logical system architecture, and functional descriptions of each logical subsystem component.

Chapter 4 – Advertisement and Discovery: Captures design for AllJoyn service advertisement and discovery, including architecture, message flows and message structure.

Chapter 5 – AllJoyn Session: Captures design for AllJoyn session, including architecture, message flows and over-the-wire interface.

Chapter 6 – Data Exchange: Captures design for data exchange between AllJoyn applications via method calls, signals and properties.

Chapter 7: Sessionless Signal: Captures design for sessionless signals, including architecture, message flows and over-the-wire interface.

Chapter 8: AllJoyn Security: Captures AllJoyn security design including app-to-app level authentication and message encryption.

Chapter 9: Thin Application: Captures design for AllJoyn thin application, including architecture, connectivity with the AJ Router and compatibility aspects.

Chapter 10: Events and Actions: Captures design for the Events and Actions core feature.

1.3 Release history

Release version	Date	Description
14.06	9/26/2014	Initial release

1.4 References

Reference Number	Document Name	Document Number	Document Path
1	D-Bus Spec		http://dbus.freedesktop.org/doc/dbus-specification.html
2	*Introduction to the AllJoyn™ Framework*		https://allseenalliance.org/docs-and-downloads/documentation/introduction-alljoyn-framework
3	*AllJoyn™ About Service HLD*		
4	*AllJoyn™ Core Security Enhancement HLD* for release 14.06		
5	Domain Names – Implementation and Specification	RFC 1035	https://www.ietf.org/rfc/rfc1035.txt
6	The Transport Layer Security (TLS) protocol	RFC 5246	http://www.rfc-base.org/txt/rfc-5246.txt
7	Using the Secure Remote Password (SRP) Protocol for TLS Authentication	RFC 5054	http://www.rfc-editor.org/rfc/rfc5054.txt
8	Multicast DNS	RFC 6762	http://tools.ietf.org/html/rfc6762
9	DNS-based Service Discovery	RFC 6763	http://tools.ietf.org/html/rfc6763
10	A DNS RR for specifying the location of services (DNS SRV)	RFC 2782	https://www.ietf.org/rfc/rfc2782.txt

1.5 Acronyms and terms

Term	Definition
Action	A function performed by an AllJoyn-enabled device.
Action-receiving device	The device that performs an action.
AJ	AllJoyn. This acronym is used in diagrams when it’s necessary to abbreviate the AllJoyn term.
AJSCL	AllJoyn Standard Core Library.
AJTCL	AllJoyn Thin Core Library.
AllJoyn application	An application that either provides an AllJoyn service or consumes an AllJoyn service. Furthermore, the application can incorporate either AllJoyn standard core library or AllJoyn thin core library.

Term	Definition
AllJoyn-enabled device	An entity which has an AllJoyn application installed to send or receive notifications using the Notification service framework interface.
AllJoyn framework	Open source peer-to-peer framework that allows for abstraction of low-level network concepts and APIs.
AllJoyn service	A service provided over the AllJoyn platform.
Authoring app	Application that carries out the IFTTT rules.
Consumer	AllJoyn application consuming services over the AllJoyn platform.
DNS	Domain Name System
DNSPTR	DNS pointer resource record
DNS-SD	DNS Service Discovery
DTIM	Delivery Traffic Indication Message
ECDHE	Elliptic Curve Diffie-Hellman Ephemeral
Event	A message denoting that something has happened.
Event-emitting device	The device that sends the event.
Event Picker app	Application that lets end users program actions to take when an event is sent.
GUID	Globally Unique Identifier. A 128 bit identifier generated randomly in a way that the probability of collision is negligible.
IANA	Internet Assigned Numbers Authority
IFTTT	If This Then That. A logical construct that tests for a certain condition and then performs an action if it is "true".
IoE	Internet of Everything
IoE device	A device that that is connected to Internet directly or via Gateway. For the purpose of this document it implies the device that has an AllJoyn application running on it.
mDNS	Multicast DNS
NAT	Network Address Translation
NGNS	Next Generation Name Service
Producer	AllJoyn application providing services over the AllJoyn platform.
Proximal network	Refers to a network that does not include a cloud-based service
Proximal IoE network	Refers to a network that includes a cloud-based service
SASL	Simple Authentication and Security Layer
Security	Framework for AllJoyn applications to authenticate each other and send encrypted data between them.
SLS	Sessionless Signal.
SRP	Secure Remote Password
Thin app	An AllJoyn application that incorporates the AllJoyn thin core library.
WKN	Well-Known Name. This acronym is used in diagrams when it’s necessary to abbreviate the Well-Known Name term.

2 AllJoyn System Overview

2.1 IoE overview

The Internet of Everything (IoE) is an exciting vision which promises to connect people with things and things with each other in ways; this will create new capabilities and richer experiences and overall make our life simpler. IoE promises to bring people, process, data, and things together to make networked connections more relevant and valuable than ever before, turning information into actions and enabling capabilities never possible before.

The IoE will result in smart things or devices at homes, offices, cars, streets, airports, malls etc. and these devices will work together to provide contextual and real time experiences to users. The IoE devices nearby each other will form proximal IoE networks e.g., at home, in the car, or in one's office. The IoE vision will enable internetworking among multiple proximal IoE networks.

It is worthwhile to draw a comparison between Internet today and Internet of Everything. The Internet today consists of millions of registered top-level domains names centrally managed by the Internet Assigned Numbers Authority (IANA). Discovery of these domains happen through a hierarchical lookup via the Domain Name System (DNS). In an IoE network, there will be potentially tens of billions of IoE devices. It would be impossible to manage registration for all of these devices via a central entity from a scalability perspective. Also, in an IoE network, proximity-based interactions among devices reduces latency as well as the need for each device to connect directly to the Internet. So, the discovery should happen automatically based on proximity criteria. Security and privacy issues become even more important in an IoE network where more and more personal and home devices expose interfaces for connection and control.

Figure 2-1 shows an example IoE network with multiple proximal IoE networks interconnected with each other via the Internet. Smart devices within each IoE proximal network can dynamically discover and communicate with each other over direct peer-to-peer connection. For cases where some of these devices are behind NAT, they can discover each other via some cloud-based discovery services. The cloud-based discovery can also be used for discovery and connection among IoE devices in different IoE proximal networks. The overall IoE network can have additional cloud-based services to provide specific functionality e.g., remote home automation, remote diagnostics/maintenance, data collection/reporting etc. The IoE network can also integrate with some of the existing cloud-based services e.g., integration with Facebook or Twitter for device status updates.

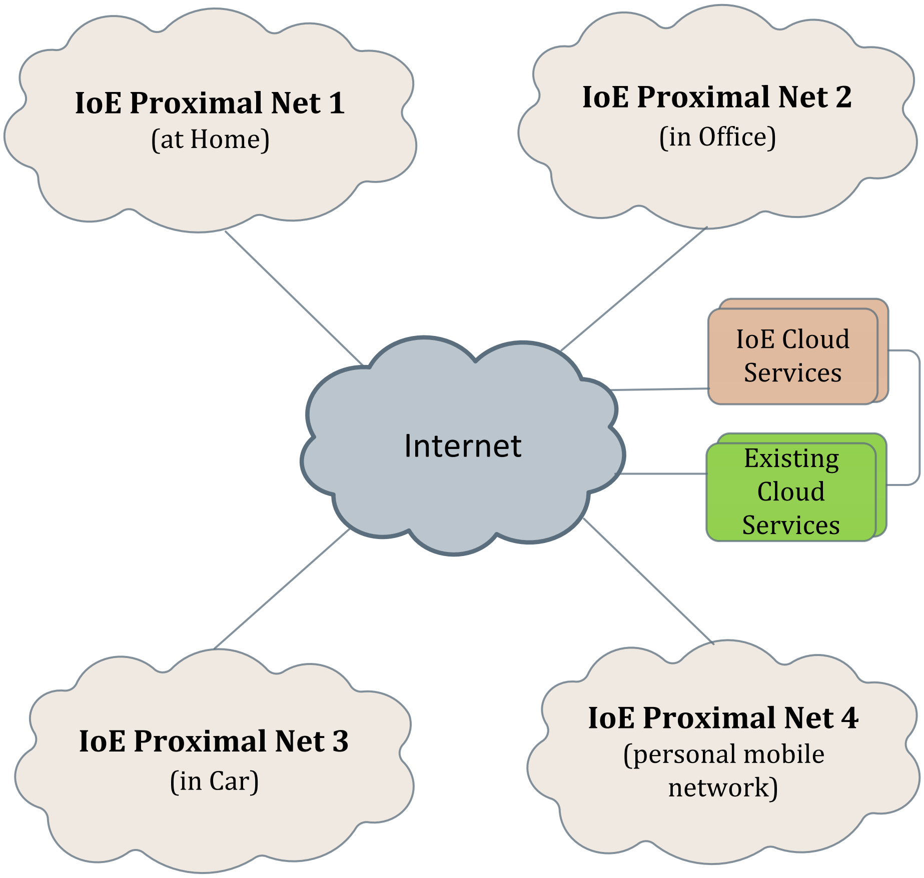


Figure 2-1. IoE network example

In any IoE network, interoperability among devices within and across IoE proximal networks is of utmost importance to enable a rich, scalable IoE ecosystem of applications and services for these devices. Any IoE system must consider specific key design aspects including device advertisement and discovery, mobility and dynamic IoE network management, security and privacy, interoperability across multiple bearers/OS, lightweight solution to support thin/dumb device, extensibility and overall scalability. For an IoE system to be truly adapted and successful, it must be open and provide a horizontal solution that can be used across different vertical use cases.

The AllJoyn system aims to address these key design aspects. It provides an open-source software framework to enable proximity-based, peer-to-peer, bearer-agnostic networking among IoE devices. The AllJoyn system provides a way for devices and applications to advertise and discover each other using peer-to-peer protocol within a proximal network.

2.2 AllJoyn system overview

The AllJoyn open-source software system provides a framework for enabling communication among IoE devices across heterogeneous distributed systems. The AllJoyn system is a proximity-based, peer-to-peer communication platform for devices in a distributed system. It does not require a centralized server for communication across such devices. AllJoyn-enabled devices run one or more AllJoyn applications and form a peer-to-peer AllJoyn network. The AllJoyn system is a distributed software platform which enables applications running on IoE devices to advertise, discover and connect to each other for making use of services offered on these devices. The AllJoyn framework enables these applications to expose their functionality over the network via discoverable APIs which are the contracts that define the functionality provided by the application.

In the proximal AllJoyn network, AllJoyn applications installed on IoE devices are peers to each other. An AllJoyn-enabled application can play the role of a provider, a consumer or both depending upon the service model. Provider applications implement services and advertise them over the AllJoyn network. Consumer applications interested in these services discover them via the AllJoyn network. Consumer applications then connect to provider applications to make use of these services as desired. An AllJoyn application can act as both provider and consumer at the same time. This means that the app can advertise a certain set of services it supports, and can also discover and make use of services provided by other apps in the proximal AllJoyn network.

Figure 2-2 shows an AllJoyn network with 4 devices. Device 1 and Device 2 have only Provider applications providing AllJoyn services. Device 3 has only consumer applications consuming services from other provider devices. Device 4 has an application which acts as both provider and consumer. The application on Device 4 consumes services from the application on Device 2. It also provides services which get consumed by applications on Device 3. Arrow directions are from provider to consumer indicating consumption of services.

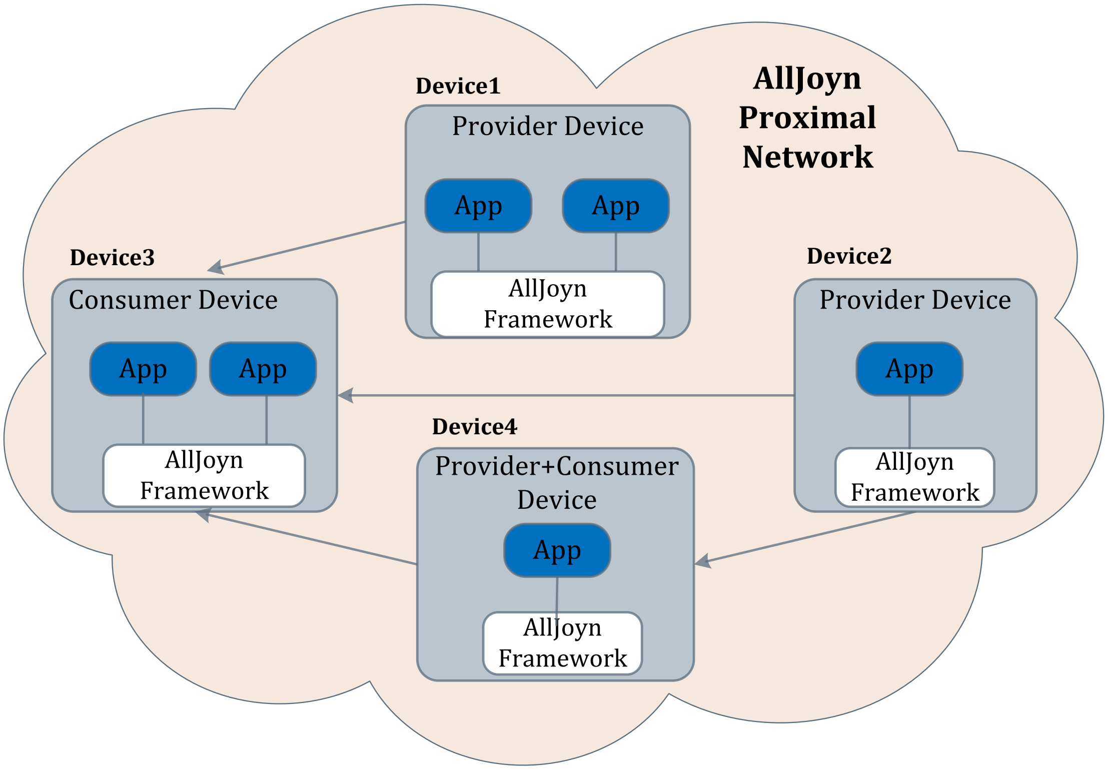


Figure 2-2. AllJoyn network

The AllJoyn framework establishes an underlying bus architecture for communication among IoE devices. AllJoyn applications on IoE devices connect and communicate to each other via the AllJoyn Bus. The AllJoyn bus provides a framework for applications to expose their services to other AllJoyn applications. The AllJoyn bus provides a platform- and radio-link agnostic transport mechanism for applications on IoE devices to send notifications or exchange data. The AllJoyn bus takes care of adapting to an underlying physical network-specific transport.

Each AllJoyn app connects to a local AllJoyn bus. One or more applications can connect to a given local AllJoyn bus. AllJoyn bus enables attached AllJoyn applications to advertise, discover, and communicate with other. AllJoyn buses on multiple devices communicate with each other using underlying network technology such as Wi-Fi.

The AllJoyn framework's open-source implementation provides an ecosystem where various parties can contribute by adding new features and enhancements to the AllJoyn system. It supports OS independence via an OS abstraction layer allowing the AllJoyn framework and its applications to run on multiple OS platforms. The AllJoyn framework supports most standard Linux distributions, Android 2.2 and later, common versions of Microsoft Windows OS, Apple iOS, Apple OS X and embedded OSs such as OpenWRT and RTOSs like ThreadX.

The AllJoyn framework also supports multiple programming languages for writing applications and services for IoE devices, which enable a wide ecosystem for developing AllJoyn applications and services. The AllJoyn framework currently, supports C, C++, Java, C#, JavaScript, and Objective-C.

2.3 The AllJoyn system and D-bus specification

The AllJoyn system implements a largely compatible version of the D-Bus over-the-wire protocol and conforms to many of the naming conventions and guidelines in the D-Bus specification. The AllJoyn system has extended and significantly enhanced D-Bus message bus to support a distributed bus scenario. The AllJoyn system makes use of the D-Bus specification as follows:

- It uses the D-Bus data type system and D-Bus marshaling format.
- It implements an enhanced version of the D-Bus over-the-wire protocol by adding new flags and headers (detailed in section 6.8).
- It uses D-Bus naming guidelines for naming well-known names (Service names), interface names, interface member names (methods, signals and properties) and object path names.
- It uses a D-Bus defined Simple Authentication and Security Layer (SASL) framework for application layer authentication between AllJoyn-enabled applications. It supports authentication mechanisms beyond what are defines by the D-Bus specification.

The D-Bus specification can be found at http://dbus.freedesktop.org/doc/dbus-specification.html.

2.4 AllJoyn system key concepts

As previously stated, the AllJoyn framework provides an underlying bus architecture for applications to advertise, discover, and make use of each other's functionality. To achieve this, the AllJoyn framework provides an object-oriented software framework for applications to interact with each other.

This section captures some of the key concepts of the AllJoyn system.

2.4.1 AllJoyn router

The AllJoyn Router component provides core functionality of the AllJoyn system, including peer-to-peer advertisement/discovery, connection establishment, broadcast signaling and control/data messages routing. The AllJoyn router implements software bus functionality and an application connects to this bus to avail core functions of the AllJoyn framework. Each instance of the AllJoyn router has an associated globally unique identifier (GUID) which is self-assigned. Currently, this GUID is not persisted, so a new GUID is assigned whenever the AllJoyn router starts up.

An AllJoyn router can be either bundled with each application (bundled model), or can be shared across multiple applications (standalone model) on the device as shown in Figure 2-3.

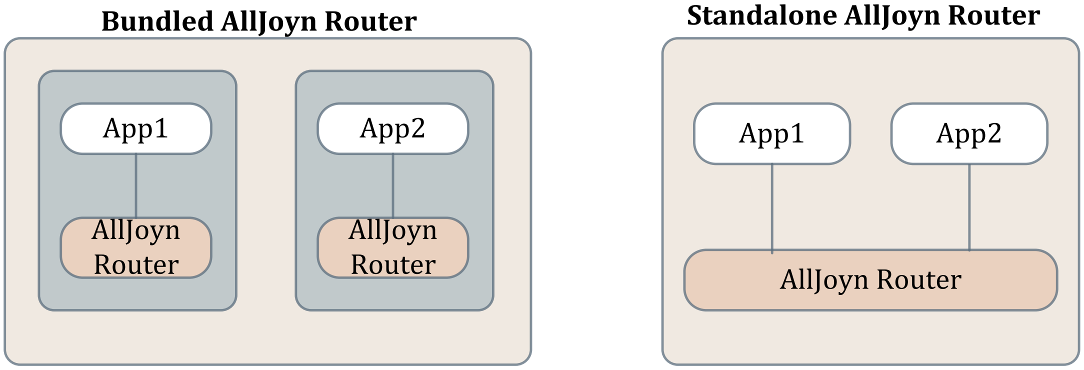


Figure 2-3. AllJoyn bundled and standalone router examples

An AllJoyn router has an associated AllJoyn protocol version that defines the set of functionality it supports. This protocol version is exchanged between AllJoyn routers on an AllJoyn network when they establish connection with each other as part of AllJoyn session establishment.

2.4.2 AllJoyn bus

An AllJoyn router provides software bus functionality where one or more applications can connect to it to exchange messages. AllJoyn router instances on a device form a logical AllJoyn bus local to the device as shown in Figure 2-4.

The logical AllJoyn bus maps to a single AllJoyn router in two cases:

- Bundled deployment model with only one app on the device, shown as UC2 in Figure 2-4.
- Standalone deployment model with one or more apps on the device, shown as UC3 in Figure 2-4.

The logical AllJoyn bus maps to multiple AllJoyn router instances in the bundled deployment model with multiple apps on the device, shown as UC1 in Figure 2-4.

NOTE

The AllJoyn router and AllJoyn bus terminology are used interchangeably in this document as these refer to same set of bus functionality provided by the AllJoyn system.

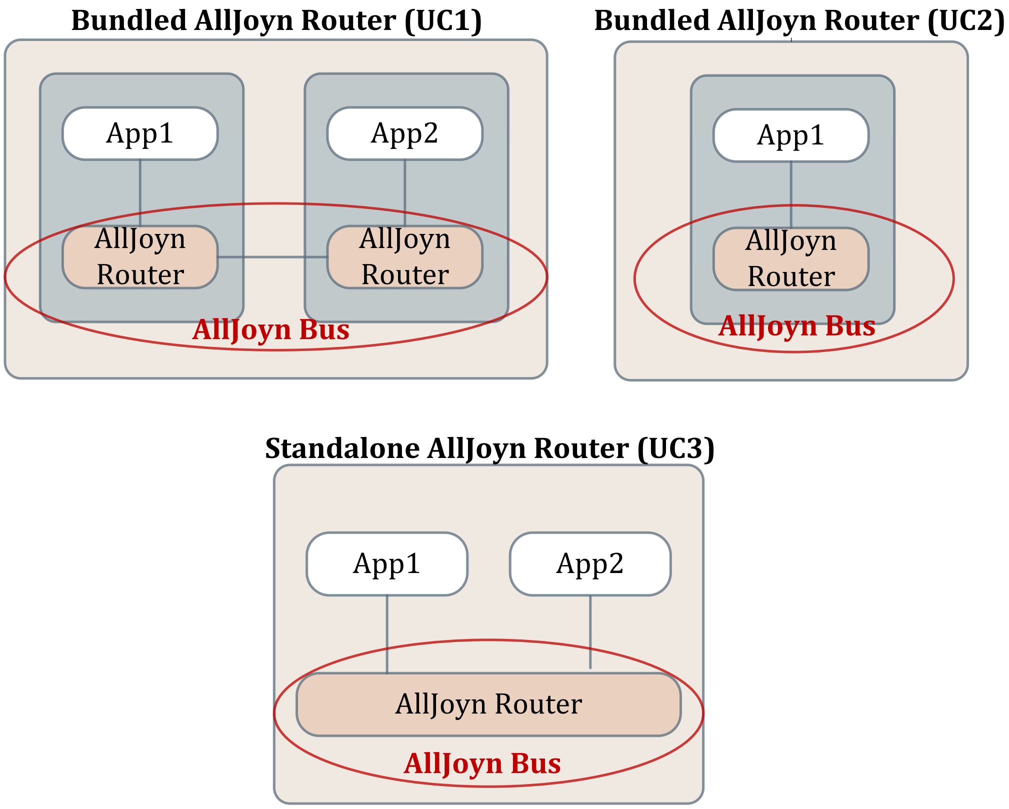


Figure 2-4. Logical mapping of AllJoyn router to AllJoyn Bus

Figure 2-5 shows a simplistic view of the local AllJoyn bus on two different devices with multiple applications connecting to the bus. The AllJoyn bus provides a medium for communication between apps connected to the bus. AllJoyn buses on multiple devices communicate with each other using the underlying network technology such as Wi-Fi.

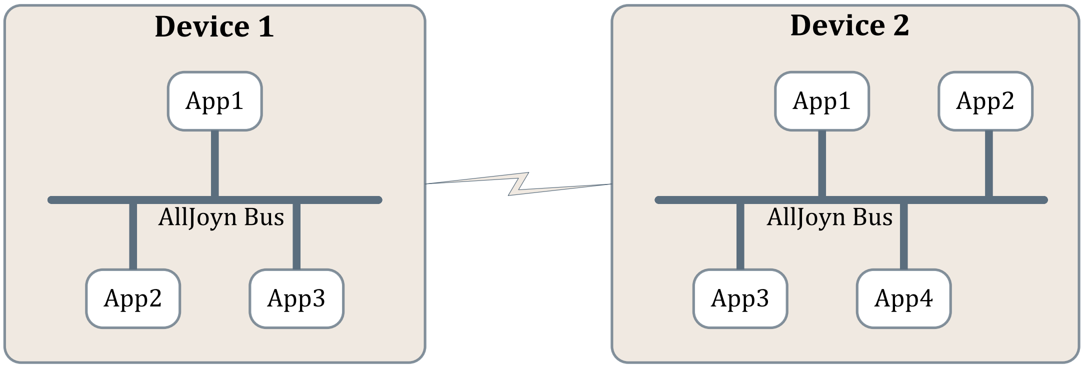


Figure 2-5. AllJoyn bus

Multiple instances of AllJoyn buses across multiple devices form a logical distributed AllJoyn software bus as shown in Figure 2-6. The distributed AllJoyn bus hides all the communication link details from the applications running on multiple devices. To an application connected to the AllJoyn bus, a remote application running on another device looks like an app that is local to the device. AllJoyn distributed bus provides a fast lightweight way to move messages across the distributed system.

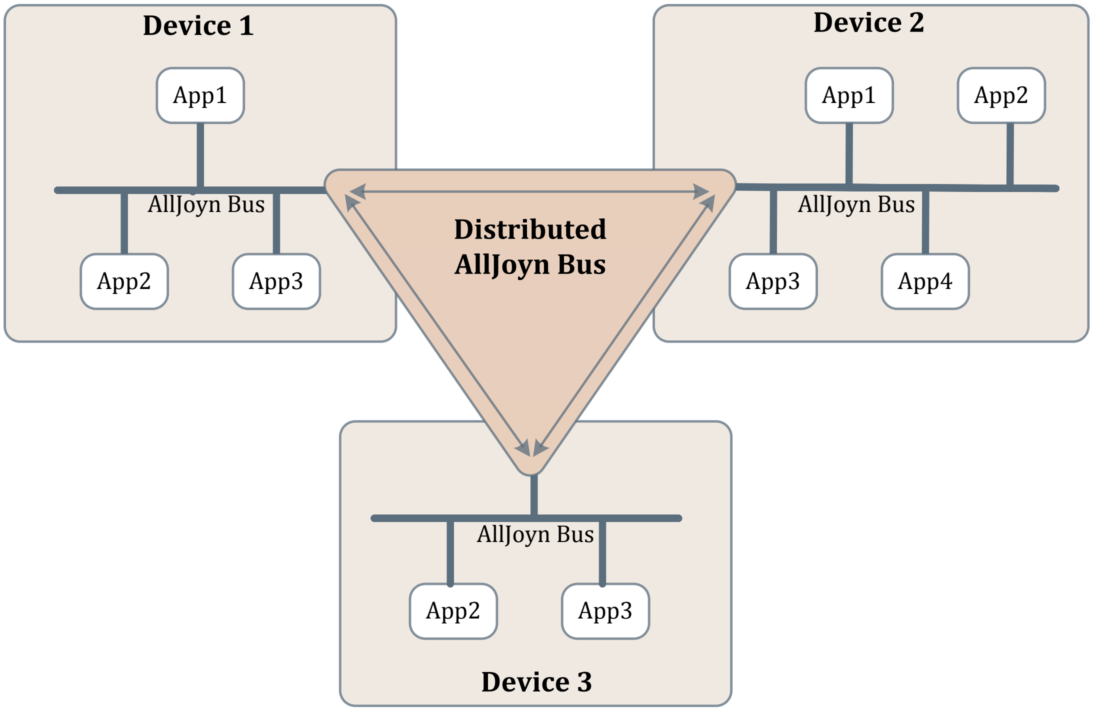


Figure 2-6. Distributed AllJoyn bus example

2.4.3 AllJoyn service

As described earlier, provider AllJoyn applications provide services that can be consumed by other applications in the AllJoyn network. For example, a TV may provide a picture rendering service to display pictures from another AllJoyn device (e.g., smartphone). An AllJoyn Service is a notional/logical concept and is defined by one or more AllJoyn interfaces (described in section 2.4.7) which expose service functionality to consumers.

An AllJoyn application can act as both provider and consumer by providing and consuming AllJoyn services at the same time.

2.4.4 Unique name

Each AllJoyn application connects to a single AllJoyn router. To enable addressing for individual applications, an AllJoyn router assigns a unique name to each connecting application. The unique name uses AllJoyn router GUID as the prefix. It follows the format below:

Unique Name = “:”<AJ router GUID>”.”<Seq #>

NOTE

The “:<AJ router GUID>.1” unique name is always given to the AllJoyn router local endpoint.

Figure 2-7 shows the unique name assignment for three connected apps to an AllJoyn bus by a single AllJoyn router with GUID=100. This scenario illustrates a device with multiple AllJoyn applications connected to a single AllJoyn router. It is expected that a large number of AllJoyn-enabled devices will be single-purpose devices (e.g., refrigerator, oven, light bulb, etc.), and will have only one application residing on the device and connecting to the AllJoyn bus. However, there can be devices where a single instance of an AllJoyn router will support multiple applications, such as a TV.

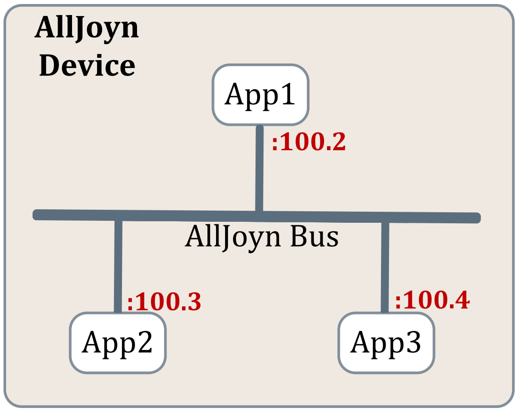


Figure 2-7. AllJoyn unique name assignment 1 (multiple apps connected to single AllJoyn router)

Figure 2-8 shows the unique name assignment for AllJoyn apps with multiple instances of an AllJoyn router forming an AllJoyn bus.

NOTE

The GUID part in each unique name is different and corresponds to the GUID for the associated AllJoyn router.

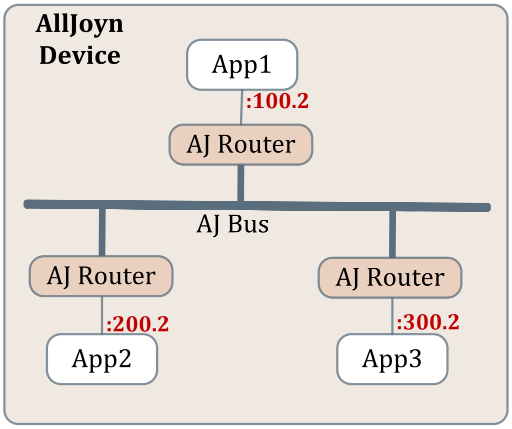


Figure 2-8. AllJoyn unique name assignment 2 (each app has instance of AllJoyn router)

Figure 2-9 shows the unique name assignment for AllJoyn apps on two different devices connected over a distributed AllJoyn bus.

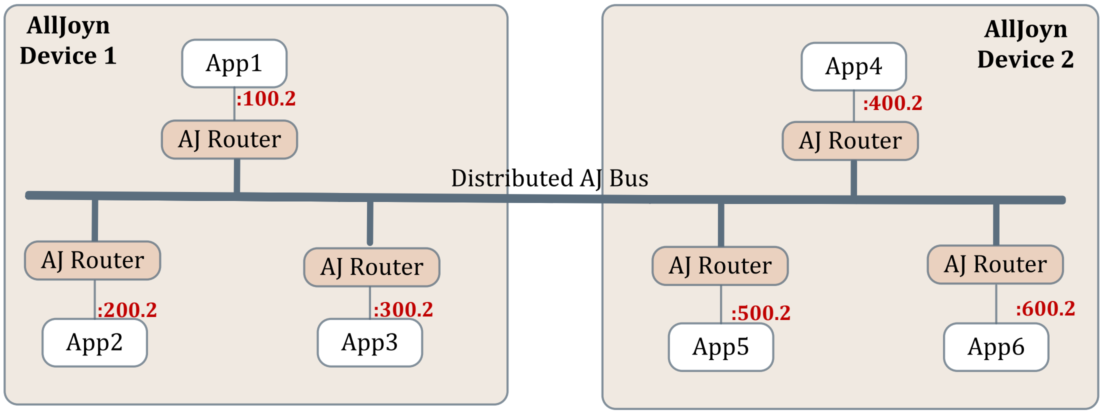


Figure 2-9. AllJoyn unique name assignment 3 (AllJoyn apps on two devices connected over distributed AllJoyn bus)

2.4.5 Well-known name

An AllJoyn application can decide to use well-known names for its services. A well-known name is a consistent way to refer to a service (or collection of services) offered over the AllJoyn bus. An app can use a single well-known name for all the services it offers, or it can use multiple well-known names across these services.

An application can request use of one or more well-known names from the AllJoyn bus for services it provides. If the requested well-known name is not already in use, exclusive use of that well-known name is granted to the application. This ensures that well-known names represent unique addresses on the AllJoyn bus at any point. The well-known name uniqueness is guaranteed only within the local AllJoyn bus. Global uniqueness for a well-known name should be achieved by adapting certain naming guidelines and format.

The AllJoyn well-known name follows the reverse domain name format. There can be multiple instances of a given application on a distributed AllJoyn bus, for example, the same refrigerator application running on two different refrigerators from the same vendor in the proximal network (one in the kitchen and one in the basement). To distinguish multiple instances of a given app on the AllJoyn bus, the well-known name should have a unique app specific identifier as a suffix, e.g., a GUID identifying the app instance.

The AllJoyn well-known name (WKN) follows the D-Bus specification guidelines for naming and has following format:

WKN = <reverse domain style name for service/app>"."<app instance GUID>

For example, a refrigerator service can use the following well-known name:

com.alljoyn.Refrigerator.12345678

2.4.6 AllJoyn object

AllJoyn applications implement one or more AllJoyn objects to support AllJoyn services functionality. These AllJoyn objects are called service objects and are advertised over the AllJoyn bus. Other AllJoyn applications can discover these objects from the AllJoyn bus and access them remotely to consume services provided by them.

A consumer application accesses an AllJoyn service object through a proxy object. A proxy object is a local representation of a remote service object that is accessed through the AllJoyn bus.

Figure 2-10 shows the distinction between the AllJoyn service object and proxy object.

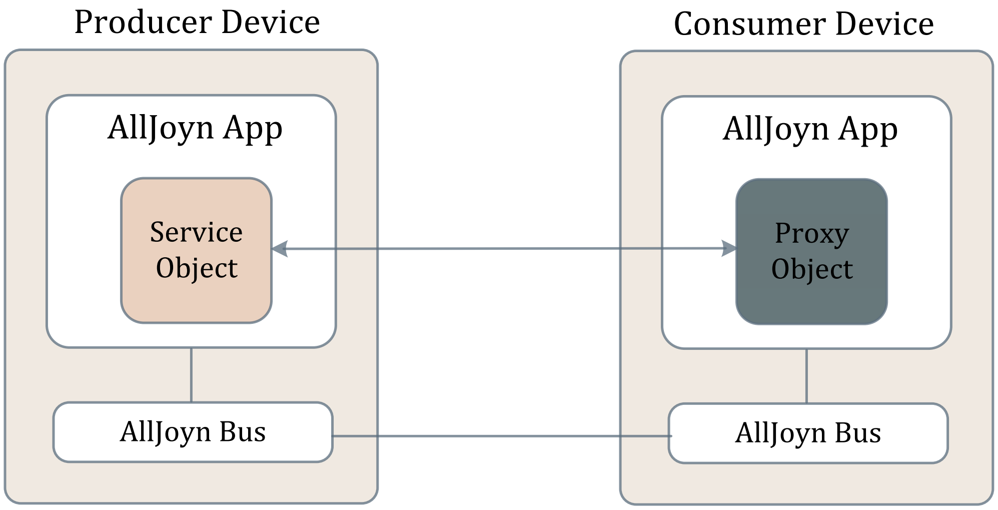


Figure 2-10. AllJoyn service object and AllJoyn proxy object

Each AllJoyn service object instance has an associated object path that uniquely identifies that object instance. This object path gets assigned when a service object gets created on the provider. The proxy object requires an object path to establish communication with the remote service object. The object path scope is within a given application, so object paths must be unique only with the associated application implementing the objects. Hence, object path naming does not need to follow reverse domain naming convention, and it can be of any form chosen by the application.

The object path naming also adheres to the D-Bus specification naming guidelines. An example object path for the service object implemented by a refrigerator can be:

/MyApp/Refrigerator

2.4.7 AllJoyn interfaces

Each AllJoyn object exposes its functionality over the AllJoyn bus through one or more AllJoyn interfaces. An AllJoyn interface defines a contract for communication between an entity implementing the interface specification and other entities interested in making use of the services provided by the interface. The AllJoyn interfaces are candidates for standardization to enable inter-operability among AllJoyn enabled IoE devices.

An AllJoyn interface can include one or more of following types of members:

- Methods: A method is a function call that typically takes a set of inputs, performs some processing using the inputs, and typically returns one or more outputs reflecting the results of the processing operation. Note that it is not mandatory for methods to have input and/or output parameters. It is also not mandatory for methods to have a reply.
- Signals: A signal is an asynchronous notification that is generated by a service to notify one or more remote peers of an event or state change. Signals can be delivered over an already-established peer-to-peer AllJoyn connection (AllJoyn session), or they can be broadcast globally to all AllJoyn peers over the distributed AllJoyn bus. Signals can be of three types:
    - Session-specific signals: These signals get delivered to one or more peers connected over a given AllJoyn session in the proximal network. If a destination is specified, the signal is delivered to only that destination node connected over the AllJoyn session. If no destination is specified, the signal gets delivered to all nodes connected over the given session except the node that generated the signal. If the session is a multi-point session, such a signal is sent over multicast to all the other participants.
    - Session broadcast signals: These signals get delivered to all the nodes connected via any AllJoyn session in the proximal network.
    - Sessionless signals: These signals get delivered to all the nodes in a proximal network that have expressed interest in receiving sessionless signals. Nodes do not need to be connected over an AllJoyn session to receive such signals. Sessionless signals are essentially broadcast signals independent of a session connection.
- Properties: A property is a variable that holds values and it may be read-only, read-write or write-only.

Every AllJoyn interface has a globally unique interface name that identifies the grouping of methods, signals, and properties provided by that interface. The AllJoyn interface name gets defined as part of standardizing the interface. Similar to the well-known name, the AllJoyn interface name also follows reverse domain name format and D-Bus specification naming guidelines.

For example, a refrigerator could support the following standard AllJoyn refrigerator interface.

org.alljoyn.Refrigerator

2.4.8 AllJoyn core library

The AllJoyn Core Library exposes AllJoyn bus functionality to AllJoyn applications. Each application links with a single instance of AllJoyn core library to connect with the AllJoyn bus. The AllJoyn core library acts as an application's gateway for peer-to-peer communications with other remote AllJoyn apps. It can be used to connect to the bus, to advertise services, to discover services, to establish connection with remote peer, to consume services, and many other AllJoyn functions. An application registers its objects with the AllJoyn core library to advertise these over the AllJoyn bus.

Figure 2-11 shows three apps connecting to a given AllJoyn bus via the AllJoyn core Library. An AllJoyn core library can be a Standard Core Library (SCL), developed for use by AllJoyn standard applications or a Thin Core Library (TCL) developed for use by AllJoyn thin applications. Most of the system design in the document is described using standard core library deployment. For thin core library design details, see chapter 9.

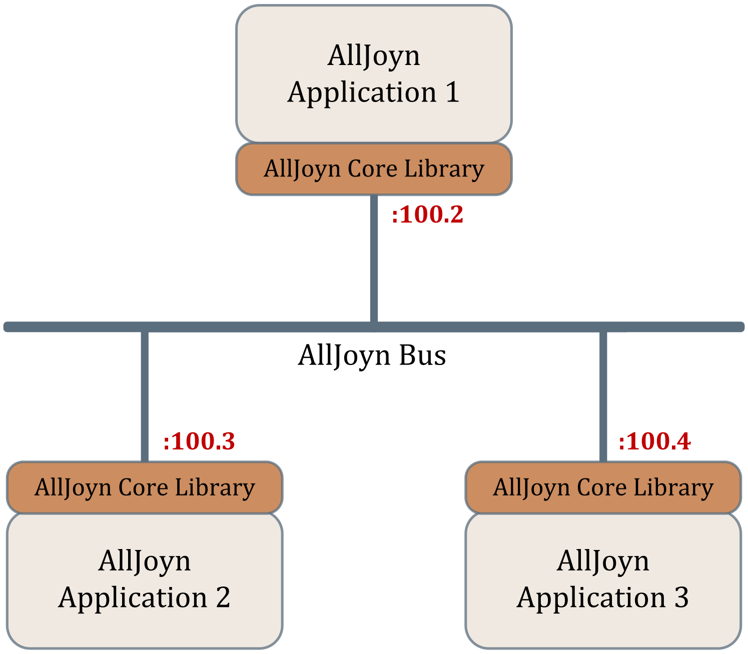


Figure 2-11. AllJoyn core library

2.4.9 About feature

The AllJoyn framework supports the About feature as part of the AJ Core Library. The About feature enables an application to expose key information about itself including app name, app identifier, device name, device identifier and a list of AllJoyn interfaces supported by the app among other details. This feature is supported by org.alljoyn.About interface implemented by the org.alljoyn.About object.

An application advertises key information about itself via an Announce signal defined by the About interface. This signal is sent as a sessionless signal on the proximal AllJoyn network. Any AllJoyn applications interested in discovering services via the AllJoyn interfaces make use of the Announce signal for discovery. The About feature also provides a mechanism to fetch application data via a direct method call. Refer to [3] for design details on the About feature.

2.4.10 AllJoyn endpoints

AllJoyn applications exchange data in the form of D-Bus formatted messages. These messages specify source and destination as endpoints. An endpoint essentially identifies a given connection with the AllJoyn router. Endpoints are used to route messages to appropriate destinations. An endpoint is uniquely identified by a unique name assigned to it.

The AllJoyn framework supports the following types of endpoints:

- **Local Endpoint:** A local endpoint is an endpoint within the AllJoyn router itself. It identifies a connection to self and is used to exchange AllJoyn control messages between AllJoyn routers. This is the first endpoint which gets assigned and always has the unique name ":<AJ router GUID>.1"
- **App Endpoint:** An app endpoint identifies the connection between the application and the AllJoyn router. Messages destined to applications get routed to app endpoints.
- **B2B Endpoint:** A bus-to-bus (B2B) endpoint identifies the connection between two AllJoyn routers. This endpoint is used as next hop to route messages between AllJoyn routers.

A routing table is maintained at the AllJoyn router that is responsible for routing messages to different types of endpoints. Control messages between two AllJoyn routers (e.g., AttachSession message) get routed to the local endpoint. AllJoyn messages between two applications get routed to app endpoints. These messages will have app endpoints as source and destination within the message. B2B endpoints are used as the next hop when routing messages (app-directed or control messages) between two AllJoyn routers.

Figure 2-12 shows different endpoints in the AllJoyn system.

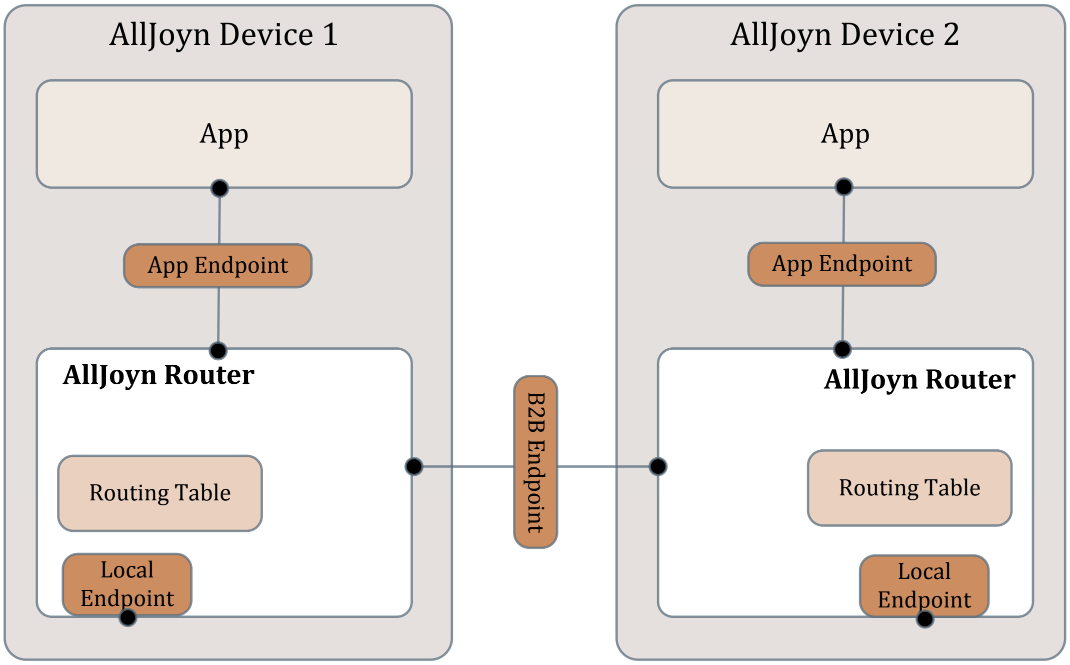


Figure 2-12. AllJoyn endpoints

2.4.11 Introspection

The AllJoyn system supports D-Bus defined introspection feature that enables AllJoyn objects to be introspected at runtime, returning introspection XML describing that object. The object should implement org.freedesktop.DBus.Introspectable interface. This interface has an Introspect method that can be called to retrieve introspection XML for the object.

2.4.12 AllJoyn entity relationship

It is useful and important to understand how different high-level AllJoyn entities relate to each other.

Figure 2-13 captures relationship between various high-level AllJoyn entities including device, application, objects, interfaces, and interface members. An AllJoyn-enabled device can support one or more AllJoyn applications. Each AllJoyn application supports one or more AllJoyn objects that implement desired application functionality. Application functionality can include providing AllJoyn services or consuming AllJoyn services, or both. Accordingly, objects supported by the AllJoyn application can be service objects, proxy objects, or combination of both. A service object exposes its functionality via one or more AllJoyn interfaces. Each AllJoyn interface can support one or more of methods, signals, and properties.

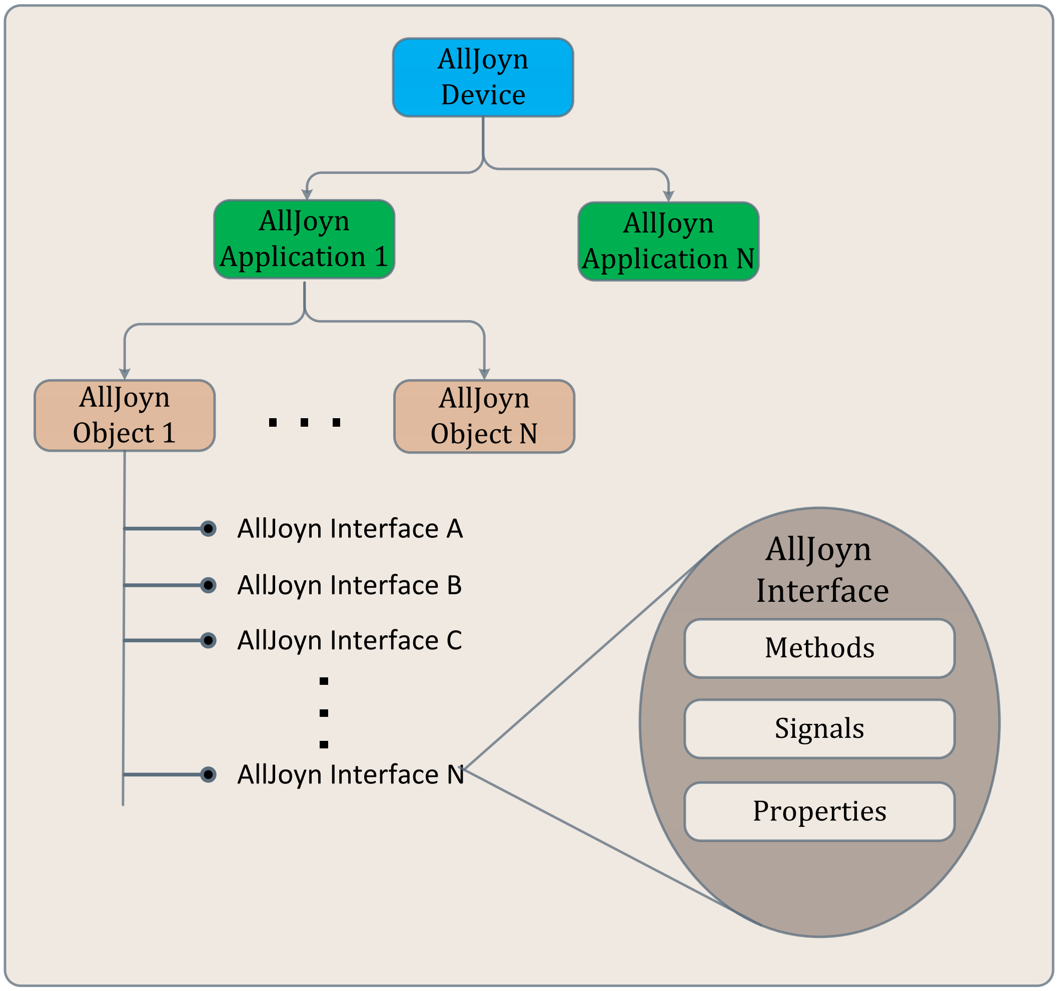


Figure 2-13. AllJoyn entity relationship

An AllJoyn service is implemented by one or more AllJoyn service objects. An AllJoyn service object can implement functionality for one or more AllJoyn services. Hence, AllJoyn service and AllJoyn service object have an n:n relationship as captured in Figure 2-14.

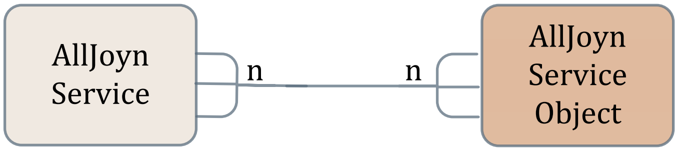


Figure 2-14. AllJoyn service-and AllJoyn service object relationship

2.5 AllJoyn services

An AllJoyn application can support one or more service frameworks and some application layer services.

2.5.1 AllJoyn service framework

AllJoyn service frameworks provide some of the core and fundamental functionality developed as enablers for higher layer application services. Service frameworks sit on top of the AllJoyn router and provide APIs to application developers to invoke their functionality. Initial AllJoyn service frameworks include Configuration service framework, Onboarding service framework, Notification service framework, and Control Panel service framework.

Service frameworks are also referred to as base services.

Example: a refrigerator application can make use of the Onboarding service framework to onboard a refrigerator to a home network and send out notifications to user devices using the Notification service framework.

2.5.2 Application layer service

An application layer service is an app-specific service provided by the AllJoyn application to achieve desired application layer functionality. These application layer services can make use of service frameworks to achieve their functionality.

Example: a refrigerator application can offer an application layer service to change refrigerator and freezer temperature. This service can make use of the Notification service framework to send out a notification when the temperature setting goes out of a specified range to notify the user.

2.6 AllJoyn transport

The AllJoyn system supports the concept of a Transport, which is used as an abstract way for an AllJoyn router to move messages from one endpoint to another. The AllJoyn transport logic in turn supports transmitting messages over multiple underlying physical transports including TCP Transport (for Wi-Fi), Bluetooth Transport, Local Transport (e.g., UNIX domain sockets) and ICE Transport.

The AllJoyn transport logic delivers the advertisement and discovery messages over multiple underlying transports based on specified list of transports by the app. Similarly, the AllJoyn transport enables session establishment over multiple underlying transports based on list provided by the application. The set of underlying transports supported by the AllJoyn transport is specified by a TransportMask which is defined in Table 2-1.

NOTE

If an app does not specify any transport(s), the AllJoyn transport value defaults to TRANSPORT_ANY.

Table 2-1. AllJoyn TransportMask definition

Transport name	Value	Description
TRANSPORT_NONE	0x0000	No transport.
TRANSPORT_LOCAL	0x0001	The local transport.
TRANSPORT_BLUETOOTH	0x0002	Bluetooth transport.

Transport name	Value	Description
TRANSPORT_WLAN	0x0004	Wireless local area network transport.
TRANSPORT_WWAN[1]	0x0008	Wireless wide area network transport.
TRANSPORT_LAN	0x0010	Wired local area network transport.
TRANSPORT_ICE[2]	0x0020	ICE (Interactive Connectivity Establishment) transport.
TRANSPORT_WFD[3]	0x0080	Wi-Fi Direct transport.
TRANSPORT_ANY	0xFFFF & ~TRANSPORT_WFD	Any transport except Wi-Fi Direct.

Currently, the AllJoyn system's WLAN and LAN transports are supported by a single underlying TCP transport.

Each transport establishes and maintains connectivity based on the underlying physical transport it supports. Based on the type of underlying physical transport, the actual connectivity between two nodes in an AllJoyn network can be either single-hop or multi-hop. An AllJoyn distributed bus is basically an overlay network whose topology doesn't map directly to the topology of the underlying network.

2.7 Advertisement and discovery

The AllJoyn framework provides a means for applications to advertise and discover AllJoyn services. The AllJoyn discovery protocol manages the dynamic nature of services coming in and going out of the proximal AllJoyn network and notifies AllJoyn applications of the same. The AllJoyn framework leverages an underlying transport-specific mechanism to optimize the discovery process. The AllJoyn framework makes use of IP multicast over Wi-Fi for service advertisement and discovery. The details of underlying mechanism are hidden from the AllJoyn applications.

The following sections details the ways that applications can use to advertise and discover services over the AllJoyn framework.

2.7.1 Name-based discovery

In the name-based discovery, advertisement and discovery typically happens using a well-known name. In this approach, the unique name can also be used for discovery per an application's discretion (e.g., if a well-known name was not assigned). A provider application advertises supported well-known names over the proximal AllJoyn network leveraging the underlying transport specific mechanism (IP multicast over Wi-Fi). These

[1] Not supported

[2] Not supported

[3] Not supported

well-known names get advertised as part of an advertisement message generated by the AllJoyn router.

A consumer application interested in a given well-known name can ask the AllJoyn router to begin discovering that name. When the provider app advertising that name comes in the proximity, the AllJoyn router receives the corresponding advertisement. The AllJoyn router then sends a service discovery notification to the application for the well-known name.

The advertisement message carries connectivity information back to the provider app. After discovery, the consumer app can request AllJoyn router to establish a connection with the discovered provider app for consuming the service. The AllJoyn router uses the connectivity information to connect back to the provider app.

2.7.2 Announcement-based discovery

Since AllJoyn services are ultimately implemented by one or more interfaces, service discovery can be achieved by discovering associated AllJoyn interfaces. In the announcement-based discovery, advertisement and discovery happens using AllJoyn interface names. This mechanism is intended to be used by devices to advertise their capabilities.

The provider application creates a service announcement message specifying a list of AllJoyn interfaces supported by that application. The service announcement message is delivered as a broadcast signal message using sessionless signaling mechanism (described in detail in Chapter 7).

Consumer applications interested in making use of AllJoyn services look for these broadcast service announcement messages by specifically registering its interest in receiving these announcements with AllJoyn router. When the consumer device is in the proximity of a provider, it receives the service announcement that contains the AllJoyn interfaces supported by the provider.

The AllJoyn router maintains connectivity information to connect back to the provider from which the service announcement message was received. After discovery, the consumer app can request the AllJoyn router to establish a connection with the provider app that supports the desired interfaces for consuming the service. The AllJoyn router uses connectivity information to connect back to the provider app.

2.7.3 Discovery enhancements in the 14.06 release

The AllJoyn discovery feature was enhanced in the 14.06 release to enable the discovery of devices/apps that support a certain set of interfaces in a more efficient way. The enhanced discovery is referred to as Next-Generation Name Service (NGNS). NGNS supports a multicast DNS (mDNS)-based discovery protocol that enables specifying AllJoyn interfaces in an over-the-wire discovery message. In addition, the mDNS-based protocol is designed to provide discovery responses over unicast to improve performance of the discovery protocol and minimize overall multicast traffic generated during the AllJoyn discovery process.

The presence detection mechanism for AllJoyn devices/apps has been enhanced by adding an explicit mDNS-based ping() message that is sent over unicast to determine if the remote endpoint is still alive. The ping() mechanism is driven by the application based on application logic.

2.8 AllJoyn session

Once a client discovers an AllJoyn service of interest, it must connect with the service in order to consume that service (except for the Notification service framework, which relies completely on sessionless signals). Connecting with a service involves establishing an AllJoyn session with that service. A session is a flow-controlled data connection between a consumer and provider, and as such allows the client to communicate with the service.

A provider app advertising a service binds a session port with the AllJoyn bus and listens for clients to join the session. The action of binding and listening makes the provider the session host. The session port is typically known ahead of time to both the consumer and the provider app. In the case of announcement-based discovery, the session port is discovered via the Announcement message. After discovering a particular service, the consumer app requests the AllJoyn router to join the session with the remote service (making it a session joiner) by specifying the session port and service's unique name/well-known name. After this, the AllJoyn router takes care of establishing the session between the consumer and the provider apps.

Each session has a unique session identifier assigned by the provider app (session host). An AllJoyn session can be one of the following:

- Point-to-point session: A session with only two participants—the session host and the session joiner.
- Multi-point session: A session with multiple participants—a single session host and multiple session joiners.

After session establishment, the consumer application must create a proxy object to interact with the provider app. The proxy object should be initialized with a session ID and the remote service object path. Once complete, the consumer app can now interact with the remote service object via this proxy object.

2.9 Sessionless signals

The AllJoyn framework provides a mechanism to broadcast signals over the proximal AllJoyn network. A broadcast signal does not require any application layer session to be established for delivering the signal. Such signals are referred to as sessionless signals and are broadcast using a sessionless signaling mechanism supported by the AllJoyn router.

The delivery of sessionless signals is done as a two-step process.

1. The provider device (sessionless signal emitter) advertises that there are sessionless signals to receive.
2. Any consumer devices wishing to receive a sessionless signal will connect with the provider device to retrieve new signals.

Using the sessionless signal mechanism, a provider application can send broadcast signals to the AllJoyn router. The AllJoyn router maintains a cache for these signals. The content of the sessionless signal cache is versioned. The AllJoyn router sends out a sessionless signal advertisement message notifying other devices of new signals at the provider device. The sessionless signal advertisement message includes a sessionless signal-specific well-known name specifying the version of the sessionless signal cache.

The consumer app interested in receiving the sessionless signal performs discovery for the sessionless signal-specific well-known name. The AllJoyn bus on the consumer maintains the latest sessionless signal version it has received from each of the provider AllJoyn router. If it detects a sessionless signal advertisement with an updated sessionless signal version, it will fetch new set of sessionless signals and deliver them to the interested consumer applications.

2.9.1 Sessionless signal enhancement in the 14.06 release

The sessionless signal feature was enhanced in the 14.06 release to enable a consumer application to request sessionless signals from provider applications that support certain desired AllJoyn interfaces. The following sessionless signal enhancements were made:

- The sessionless signal advertised name was enhanced to add <INTERFACE> information from the header of the sessionless signal. Consumers use this to fetch sessionless signals only from those providers that are emitting signals from the <INTERFACE> it is interested in. A separate sessionless signal name is advertised one for each unique interface in the sessionless signal cache.
- A mechanism was added for the consumer app to indicate receiving Announce sessionless signal only from applications implementing certain AllJoyn interfaces.

Sessionless signals are only fetched from those providers that support desired interfaces. This improves the overall performance of the sessionless signal feature.

2.10 Thin apps

An AllJoyn Thin App is designed for use in embedded devices such as sensors. These types of embedded devices are optimized for a specific set of functions and are constrained in energy, memory and computing power. An AllJoyn thin app is designed to bring the benefits of the AllJoyn framework to embedded systems. The thin app is designed to have a very small memory footprint.

A thin AllJoyn device makes use of lightweight thin application code along with the AllJoyn Thin Core Library (AJTCL) running on the device. It does not have an AllJoyn router running on that device. As a result, the thin app must use an AllJoyn router running on another AllJoyn-enabled device, essentially borrowing the AllJoyn router functionality running on that device.

At startup, the thin application discovers and connects with an AllJoyn router running on another AllJoyn-enabled device. From that point onwards, the thin app uses that AllJoyn router for accomplishing core AllJoyn functionality including service advertisement/discovery, session establishment, signal delivery, etc. If a thin app is not able to connect to previously discovered AllJoyn router, it attempts to discover another AllJoyn router to connect to.

An AllJoyn thin app is fully interoperable with an AllJoyn standard application. It uses same set of over-the-wire protocols as a standard AllJoyn app. This ensures compatibility between the thin app and standard apps. An AllJoyn standard app communicating with a thin app will not know that it is talking to a thin app and vice versa. However, there are some message size constraints that apply to the thin app based on available RAM size.

3 System Architecture

3.1 Introduction

The AllJoyn system supports multiple network deployment scenarios. Some deployments simply have all the nodes as equivalent peer nodes, while other deployments require special purpose nodes. Some deployments may also require the presence of one or more cloud servers. This chapter captures the AllJoyn network architecture for various network deployment scenarios.

An AllJoyn-enabled device can support either a single app (e.g., coffee maker) or multiple apps (e.g., TV). In the case of multiple apps on a device, each app can have its own bundled AllJoyn router, or a standalone AllJoyn router can be shared across those apps. This chapter explains the device architecture for these device deployment scenarios.

3.2 Network architecture

The AllJoyn network architecture is dependent upon the network deployment scenario. This section captures the following deployment scenarios:

- Standalone AllJoyn network: A proximal network with a set of peer devices which could be connected over the same or different access mediums.
- Remote accessible AllJoyn network: A proximal network where services provided by devices are accessible and controllable from outside the proximal network.

3.2.1 Standalone AllJoyn network

A standalone AllJoyn network architecture is fairly simple with two or more peer nodes coming together to dynamically form an AllJoyn network. Peers can be connected over different access networks such as Wi-Fi. The AllJoyn advertisement and discovery mechanism takes care of seamlessly discovering these peers independent of the underlying transport being used. Figure 3-1 captures the network architecture for a typical standalone AllJoyn network.

NOTE

In a Wi-Fi deployment, the AllJoyn framework requires wireless isolation to be turned off at the access points to enable peer-to-peer communication.

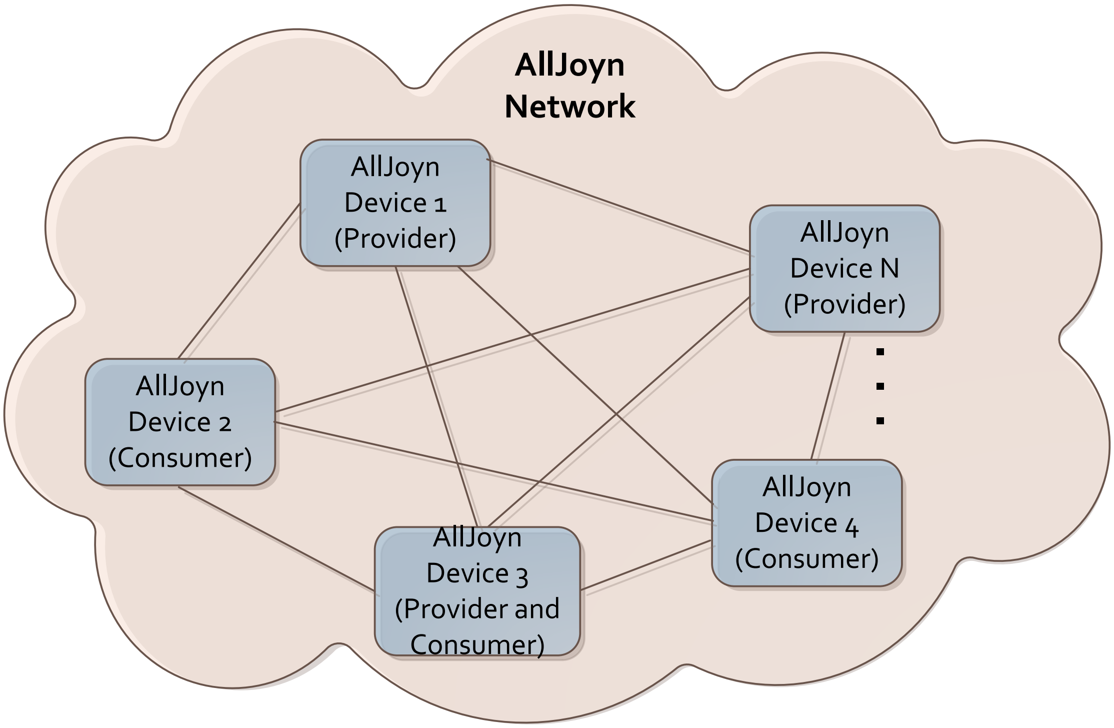

AllJoyn Network
AllJoyn Device 1 (Provider)
AllJoyn Device N (Provider)
AllJoyn Device 2 (Consumer)
AllJoyn Device 4 (Consumer)
AllJoyn Device 3 (Provider and Consumer)

Figure 3-1. Standalone AllJoyn network architecture

3.2.1.1 Bridging multiple transports

A standalone AllJoyn network can involve nodes connected over both wireless and wired transport, e.g., nodes connected over Wi-Fi, PLC, and Ethernet. Nodes in such a network can communicate with each other as long as wireless isolation is not enabled on the Wi-Fi Access Point (AP).

Figure 3-2 captures the high-level network architecture for an AllJoyn network with devices connected over Wi-Fi, PLC, and Ethernet transports.

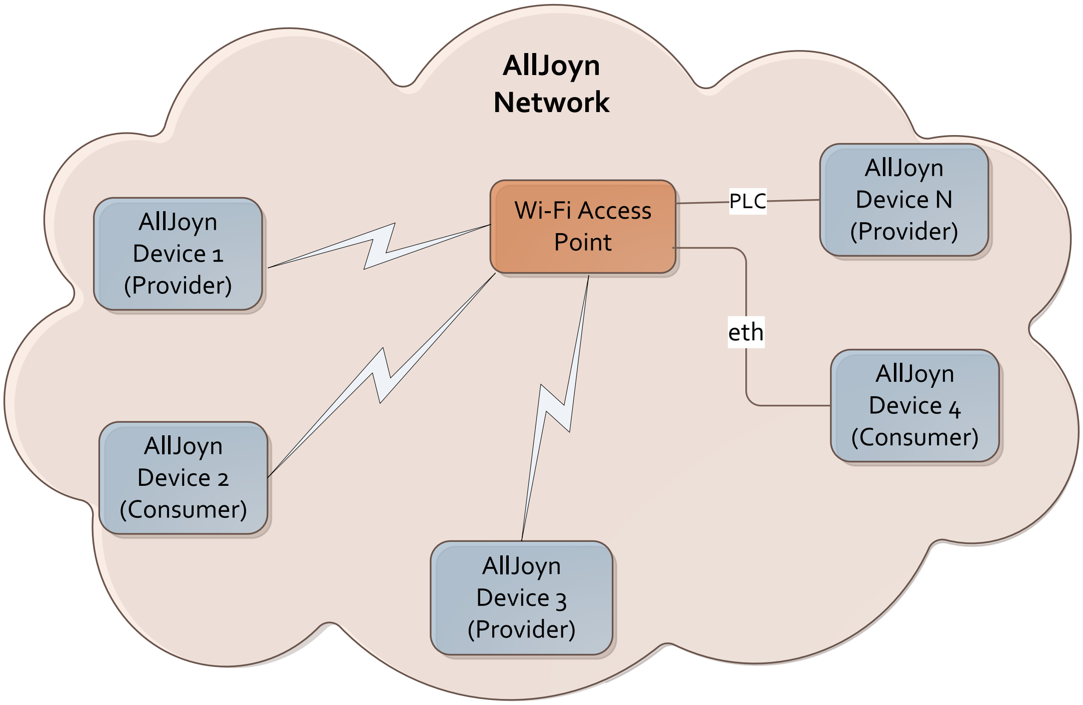

AllJoyn Network
Wi-Fi Access Point
AllJoyn Device 1 (Provider)
AllJoyn Device 2 (Consumer)
AllJoyn Device 3 (Provider)
AllJoyn Device 4 (Consumer)
AllJoyn Device N (Provider)
PLC
eth

Figure 3-2. Bridging multiple transports

3.2.2 Remote accessible AllJoyn network

A remote accessible AllJoyn network is a proximal IoE network where services provided by devices are accessible and/or controllable from outside the proximal IoE network. The remote accessibility is achieved by having a Gateway node in the system. The Gateway node exposes device functionality and control to an existing cloud-based service either via standard Internet style APIs (e.g. REST). A mobile device outside the proximal IoE network can communicate with devices in the proximal IoE network via the cloud-based service and via the Gateway node.

Figure 3-3 below captures the high-level network architecture for a remote accessible AllJoyn network.

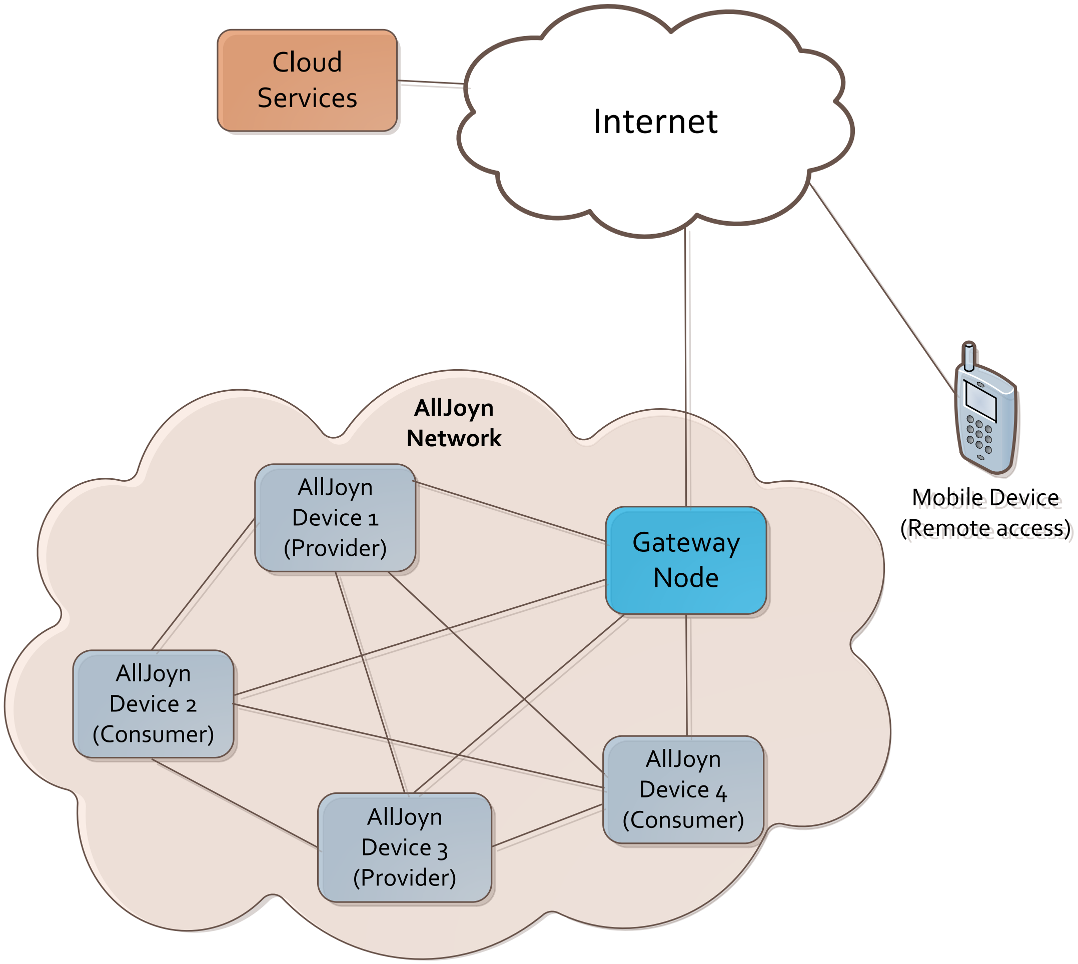

Cloud Services
Internet
Mobile Device (Remote access)
AllJoyn Network
AllJoyn Device 1 (Provider)
Gateway Node
AllJoyn Device 2 (Consumer)
AllJoyn Device 4 (Consumer)
AllJoyn Device 3 (Provider)

Figure 3-3. Remote accessible AllJoyn network architecture

3.3 Device architecture

An AllJoyn-enabled device can support one or more AllJoyn applications. The AllJoyn router can be bundled with each of these applications on devices such as mobile phone and tablets. Alternately, the AllJoyn router can be installed separately as a standalone router on the device and multiple applications can make use of it; examples of devices include TVs and set-top boxes (STBs). There can also be hybrid deployment cases where a single device has both a bundled AllJoyn router for certain apps and a standalone AllJoyn router for other apps on the device.

NOTE

An app always looks for a preinstalled AllJoyn core, so this will only happen if the preinstalled AllJoyn core was a lower version than the bundled AllJoyn core.

The following deployment scenarios are captured for the AllJoyn device:

- Single app with bundled AllJoyn router
- Multiple apps with bundled AllJoyn router

- Multiple apps with standalone AllJoyn router

3.3.1.1 Single app with bundled AllJoyn router

In this deployment, the AllJoyn application package includes an app and an AllJoyn router. The app can support application-specific services as well as one or more service frameworks. The application connects to the AllJoyn router via the AllJoyn standard core library. In case of the bundled AllJoyn router, the communication between the app and AllJoyn router is local (within the same process) and can be done using function/API calls.

Figure 3-4 captures the AllJoyn device architecture for a single app with bundled AllJoyn router deployment scenario.

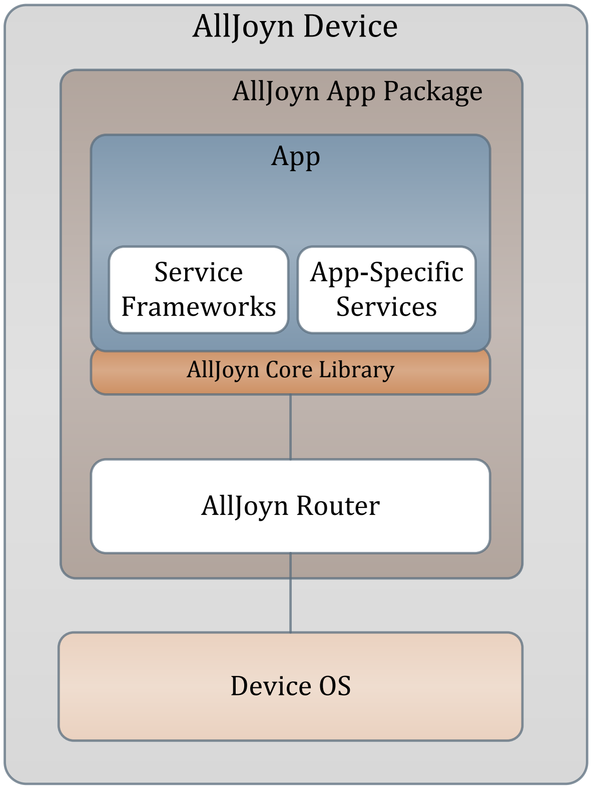


Figure 3-4. AllJoyn device architecture (single app with bundled AllJoyn router)

3.3.1.2 Multiple apps with bundled AllJoyn router

In this deployment, the AllJoyn-enabled device supports multiple applications. Each of these applications has a separate instance of an AllJoyn router bundled with that application package.

Figure 3-5 captures the AllJoyn device architecture for multiple apps with a bundled AllJoyn router deployment scenario.

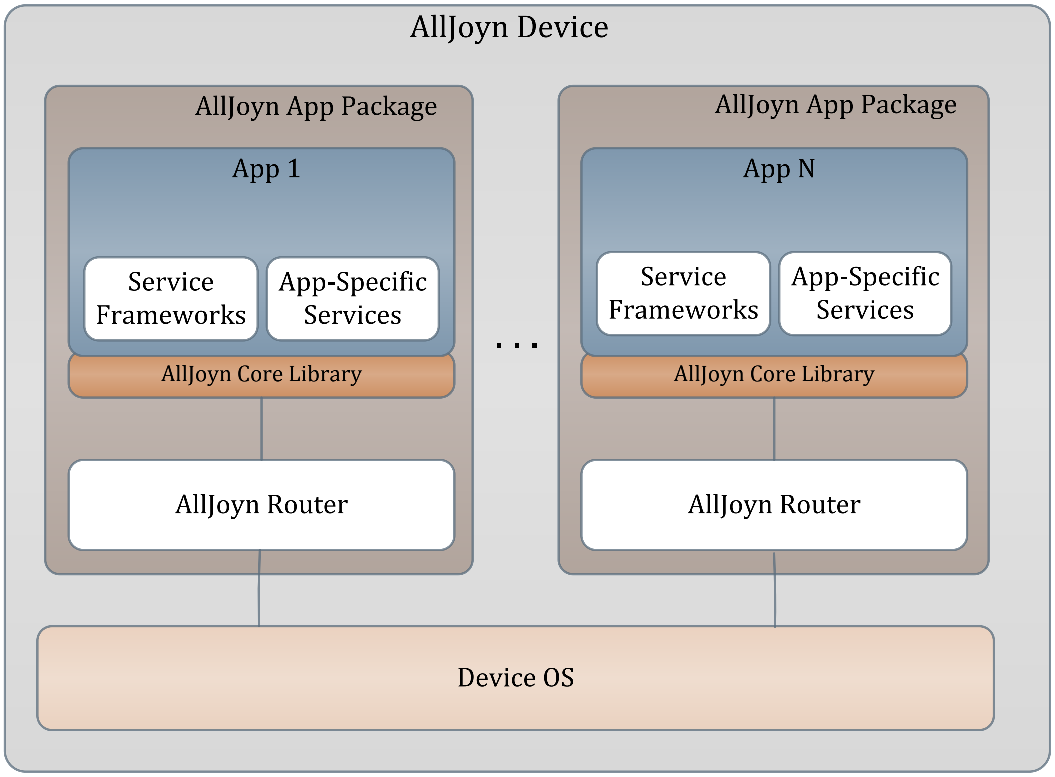


Figure 3-5. AllJoyn device architecture (multiple apps with bundled AllJoyn router)

3.3.1.3 Multiple apps with standalone AllJoyn router

In this deployment, the AllJoyn-enabled device supports a standalone AllJoyn router. The multiple applications on the device connect using the same standalone AllJoyn router. The communication between an application and the standalone AllJoyn router happens across process boundaries and can happen over transports like UNIX domains sockets or TCP.

Figure 3-6 captures the AllJoyn device architecture for multiple apps with a standalone AllJoyn router deployment scenario.

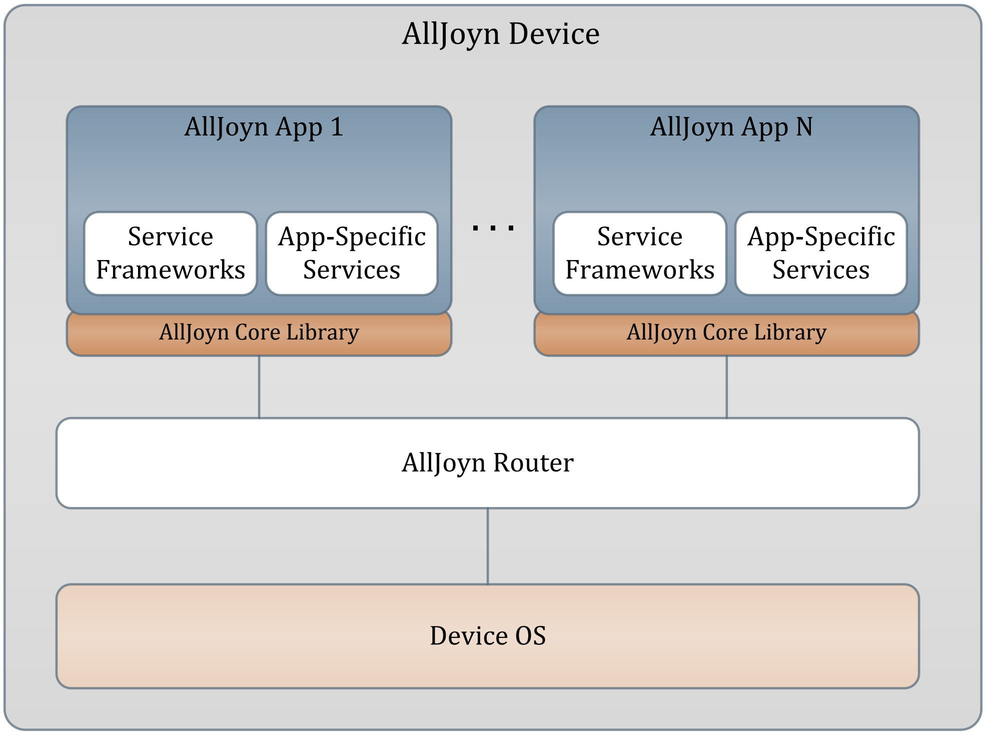


Figure 3-6. AllJoyn device architecture (multiple apps with standalone AllJoyn router)

3.4 AllJoyn router architecture

The AllJoyn router provides a number of functionalities to enable key features of the AllJoyn framework. Figure 3-7 captures the functional architecture for the AllJoyn router.

The AllJoyn router supports key features over multiple underlying transports. The Advertisement and Discovery module provides transport agnostic advertisement and discovery functionality. Similarly, modules shown for other features including Session, Data Exchange, and Sessionless Signal modules offer transport-agnostic functionality for those features. All these AllJoyn features work over various transports including Wi-Fi, wired transports, Bluetooth, and any local transport.

The AllJoyn bus management and control functions are provided by the Bus Management module. The Security module provides AllJoyn security functionality including SASL-based authentication.

The Message and Signal Transport layer provides functionality to encapsulate application layer signaling and data into D-Bus format message encapsulation. The Transport Abstraction Layer provides abstraction for various underlying transports for core AllJoyn features. The various transport-related modules provide that transport-specific functionality to accomplish core AllJoyn functions. The AllJoyn router supports an OS Abstraction Layer to interact with different underlying OS platforms.

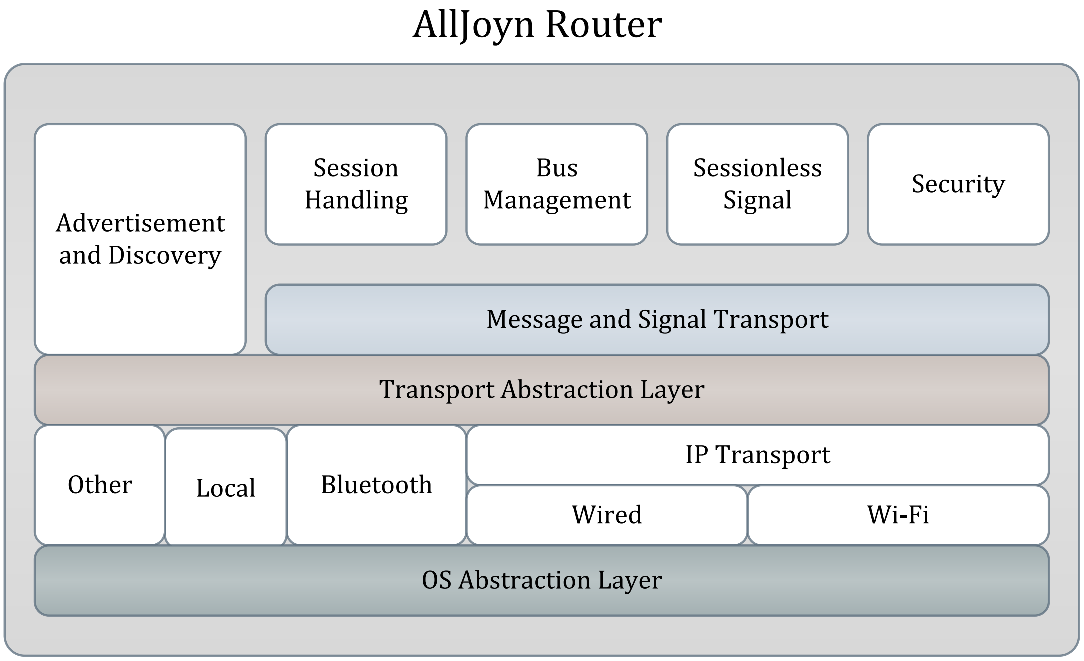


Figure 3-7. AllJoyn router functional architecture

3.5 Thin app architecture

An AllJoyn thin app is designed for energy-, memory-, and CPU-constrained devices. The thin app is designed to have a very small memory footprint and is typically single-threaded. The thin app includes the application code and AllJoyn thin core library (AJTCL); it does not include an AllJoyn router.

A thin AllJoyn device only has a lightweight thin app running on the device that makes use of an AllJoyn router running on a standard AllJoyn device to advertise, discover, and connect with AllJoyn peers. Communication between the thin app and the AllJoyn router occurs across device boundaries over TCP transport.

Figure 3-8 captures the AllJoyn thin app architecture.

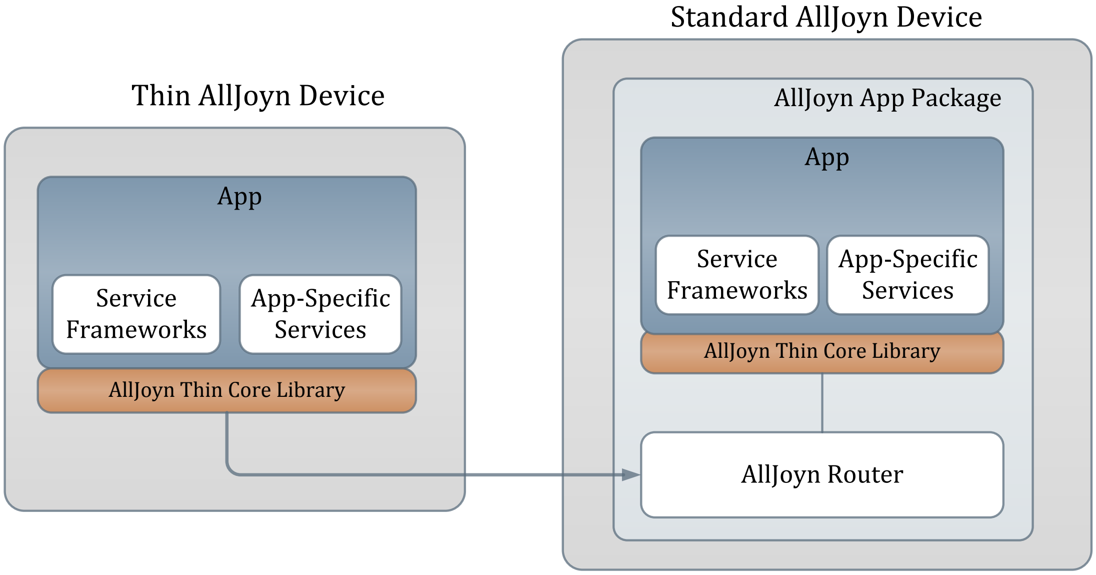


Figure 3-8. AllJoyn thin app architecture

3.6 AllJoyn framework protocol stack

Figure 3-9 captures the high-level protocol stack for the AllJoyn framework.

At the top level, the AllJoyn framework protocol stack consists of an application providing a number of application layer services and supporting some service frameworks. These app layer services are defined by AllJoyn interfaces supported by the app. The app sits on top of the AllJoyn core library, which enables an app to invoke core AllJoyn functionality.

Below the AllJoyn core library sits the AllJoyn router that implements core AllJoyn features including advertisement/discovery, session establishment, sessionless signals, authentication, etc. The AllJoyn router supports multiple underlying transports for discovery and communication and provides an abstraction layer for each of the supported transport. The AllJoyn router belongs to the application layer in the standard OSI layering model.

Under the AllJoyn router reside the standard OSI layers: transport, network, layer 2 and physical layer.

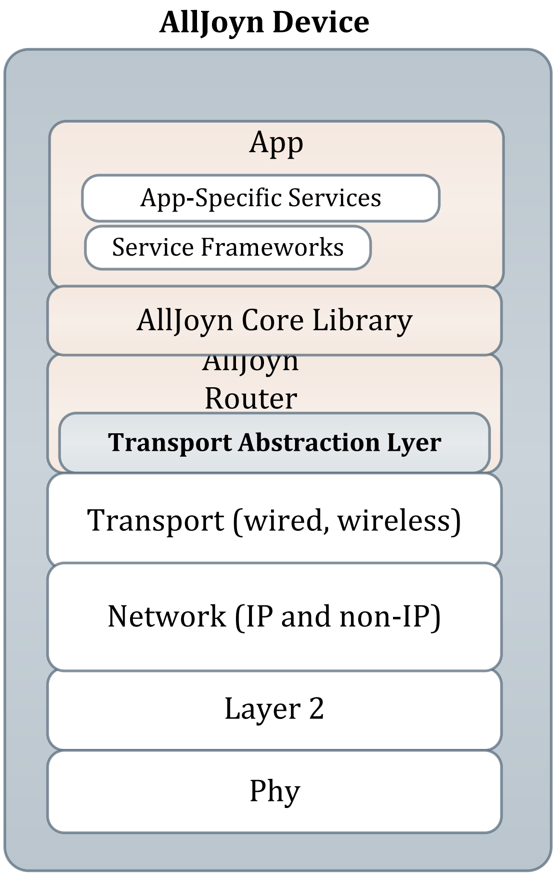


Figure 3-9. AllJoyn protocol stack

4 Advertisement and Discovery

4.1 Overview

The AllJoyn system supports a mechanism for providers to advertise their services over the AllJoyn network, and for consumers to discover these services for consumption. AllJoyn supports discovery over multiple underlying access networks such as Wi-Fi. The AllJoyn discovery protocol makes use of IP multicast over Wi-Fi for advertisement and discovery. The details of discovery over underlying networks are hidden from the AllJoyn applications.

The AllJoyn router supports transport-specific discovery for Wi-Fi using the IP transport.

Applications can use one of the following methods to advertise and discover services over the AllJoyn framework, detailed in subsequent sections of this chapter:

- Name-based discovery: Service advertisement and discovery occurs either using a well-known name or unique name.
- Announcement-based discovery: Service advertisement and discovery occurs using AllJoyn interface names.

4.1.1 Discovery enhancements in the 14.06 release

The AllJoyn discovery feature was enhanced in the 14.06 release to enable the discovery of devices/apps that support a certain set of interfaces in a more efficient way (NGNS, defined earlier). NGNS supports an mDNS-based discovery protocol that enables specifying AllJoyn interfaces in an over-the-wire discovery message. In addition, the mDNS-based protocol is designed to provide discovery responses over unicast to improve performance of the discovery protocol and minimize overall multicast traffic generated durign the AllJoyn discovery process.

The presence detection mechanism for AllJoyn devices/apps was enhanced by adding an explicit mDNS-based ping() message that is sent over unicast to determine if the remote endpoint is still alive. The ping() mechanism is driven by the application based on application logic.

This chapter first describes the legacy AllJoyn discovery (prior to the 14.06 release), followed by NGNS-based discovery and presence as described in section 4.6.

4.2 Legacy name-based discovery

This section captures design for the legacy name-based discovery supported prior to the 14.06 release.

The AllJoyn router supports a Name Service to enable the name-based service discovery. The Name Service supports a UDP-based protocol for discovery over IP-based access networks (including Wi-Fi). Name-based discovery APIs are exposed through the AllJoyn Core Library.

The Name Service supports IS-AT and WHO-HAS protocol messages, which are described below. These protocol messages carry well-known names to be advertised and discovered, respectively. These protocol messages are multicast over the AllJoyn proximal network (local subnets) over IANA-registered IP multicast groups and port number as listed in Table 4-1.

Table 4-1. IANA-registered multicast addresses for the AllJoyn framework

Address	Port
IPv4 Multicast group address	224.0.0.113
IPv6 Multicast group address	FF0X::13A
Multicast port number	9956

Figure 4-1 captures the high-level architecture for the name-based discovery, showing the Name Service generating IS-AT and WHO-HAS messages for service discovery.

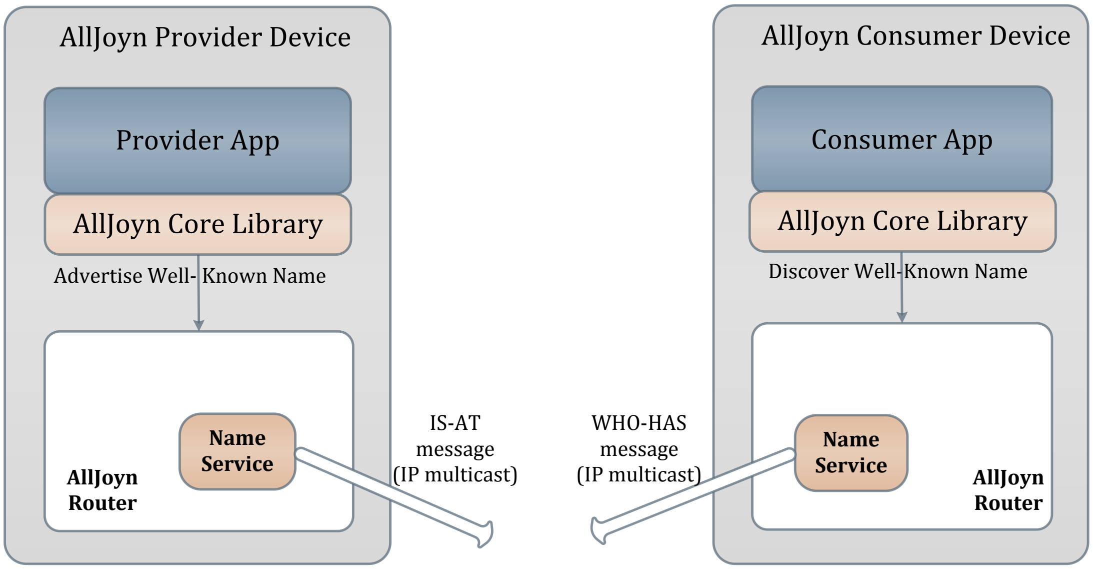


Figure 4-1. Name-based discovery architecture

4.2.1 IS-AT

The IS-AT message advertises AllJoyn services using the well-known name or the unique name. A single IS-AT message can include a list of one or more well-known names or unique names for advertisement. The IS-AT message specifies a validity period for the well-known name advertisement indicated by Adv_Validity_Period config parameter.

The AllJoyn router at the provider device send out IS-AT message periodically over IP multicast to advertise the set of services it supports. The periodicity is defined by the Adv_Msg_Retransmit_Interval period which is a configurable parameter on the provider.

The IS-AT message can also be sent out in response to a received WHO-HAS message that is looking for that advertised service. This is so that the consumer device can be immediately notified of the service to minimize the discovery time.

4.2.2 WHO-HAS

The WHO-HAS message discovers one or more AllJoyn services using the well-known name or the unique name. Similar to IS-AT, a WHO-HAS message can include a list of one or more well-known names or unique names for discovery. The WHO-HAS message can also include a well-known name prefix (instead of the complete well-known name) that gets matched against the well-known name being advertised in the IS-AT message.

For example, the well-known name prefix “org.alljyon.chat” in the WHO-HAS message match with the well-known name “org.alljoyn.chat._123456.Joe” being advertised in the IS-AT message.

When a consumer device wants to discover a service, it sends out the WHO-HAS message over IP multicast. The WHO-HAS message is repeated few times to account for the possibility of a collision on the Wi-Fi network that can result in dropping of the multicast packet.

The following parameters determine the transmission of the WHO-HAS message:

- Disc_Msg_Number_Of_Retries
- Disc_Msg_Retry_Interval

The WHO-HAS message is resent for Disc_Msg_Number_Of_Retries times at every Disc_Msg_Retry_Interval after the first transmission of the message. In response to the WHO-HAS message, a consumer can get an IS-AT message advertising the requested service from the provider.

4.2.3 Consumer behavior

Figure 4-2 captures the consumer side AllJoyn router behavior for name-based discovery.

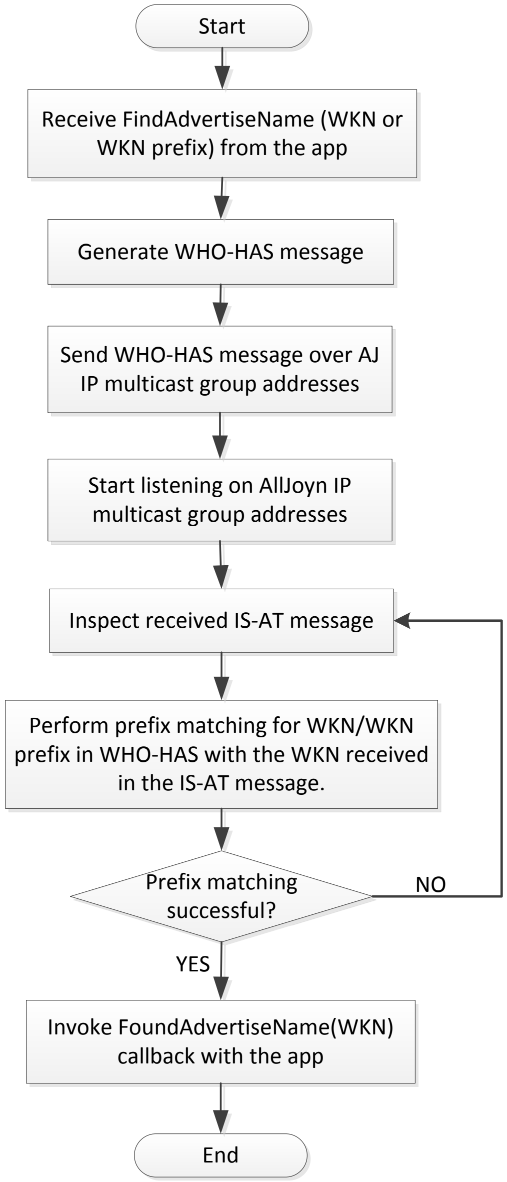

Start
Receive FindAdvertiseName (WKN or WKN prefix) from the app
Generate WHO-HAS message
Send WHO-HAS message over AJ IP multicast group addresses
Start listening on AllJoyn IP multicast group addresses
Inspect received IS-AT message
Perform prefix matching for WKN/WKN prefix in WHO-HAS with the WKN received in the IS-AT message.
Prefix matching successful?
NO
YES
Invoke FoundAdvertiseName(WKN) callback with the app
End

Figure 4-2. Consumer AllJoyn router behavior for discovery

4.2.4 Message sequence

The following use cases are captured AllJoyn name-based discovery scenarios:

- Discovery when IP connectivity is already established
- Discovery over unreliable network
- Discovery when IP connectivity is established late
- Well-known name lost due to loss of IP connectivity
- Provider cancels well-known name advertisement
- Consumer cancels discovery for well-known name

4.2.4.1 Discovery when IP connectivity is already established

Figure 4-3 shows the message sequence for a typical discovery scenario of an AllJoyn service well-known name. In this case, the provider and consumer devices already have IP connectivity established between them. The first WHO-HAS message delivered over IP multicast reaches the provider device which immediately responds with an IS-AT message.

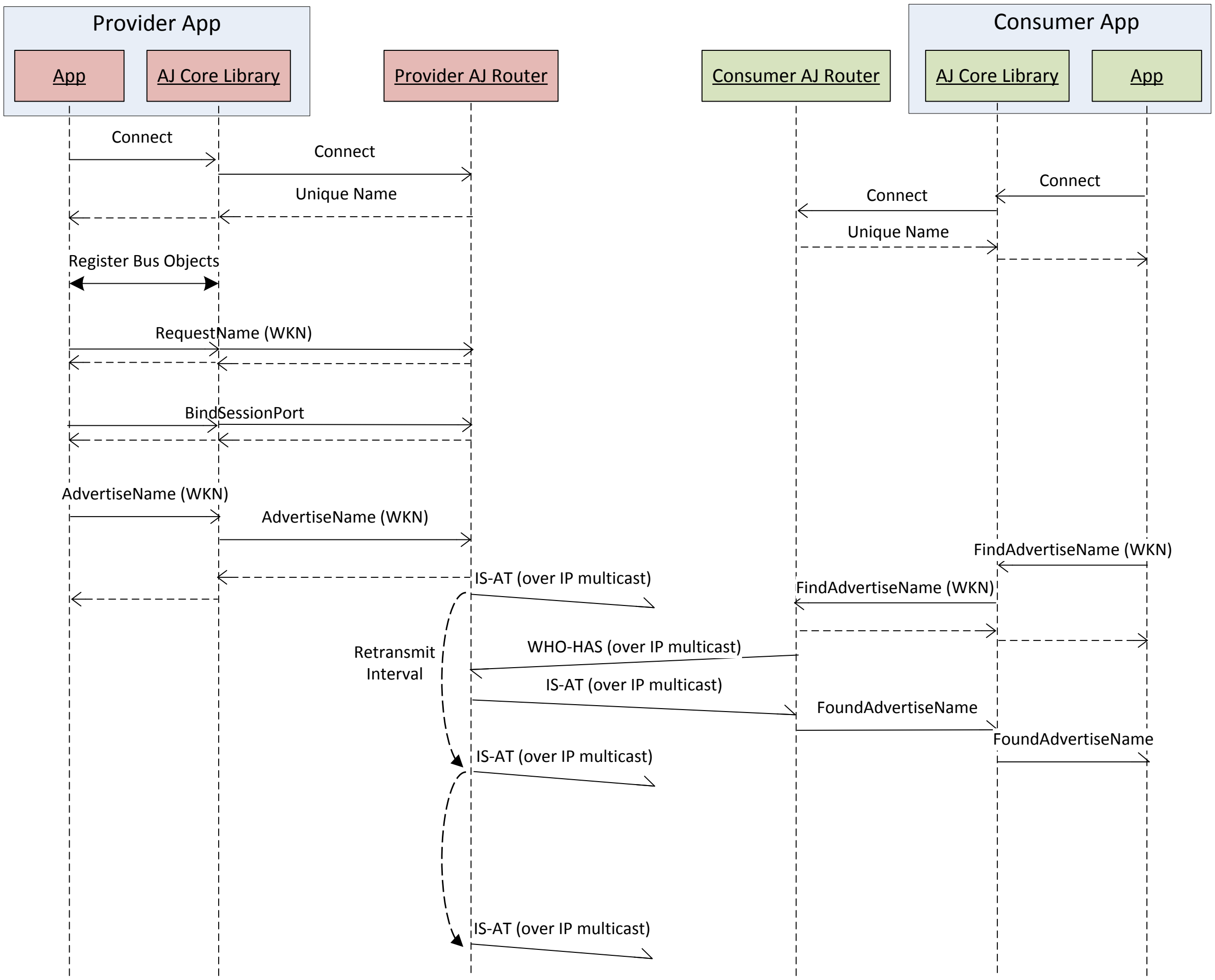


Figure 4-3. Typical discovery of a well-known name

4.2.4.2 Discovery over unreliable network

Figure 4-4 shows the message sequence for the discovery scenario of an AllJoyn service's well-known name when the underlying network drops some of the multicast WHO-HAS messages. In this case, the WHO-HAS retry mechanism kicks in and the message is retried based on Disc_Msg_Number_Of_Retries and Disc_Msg_Retry_Interval parameters.

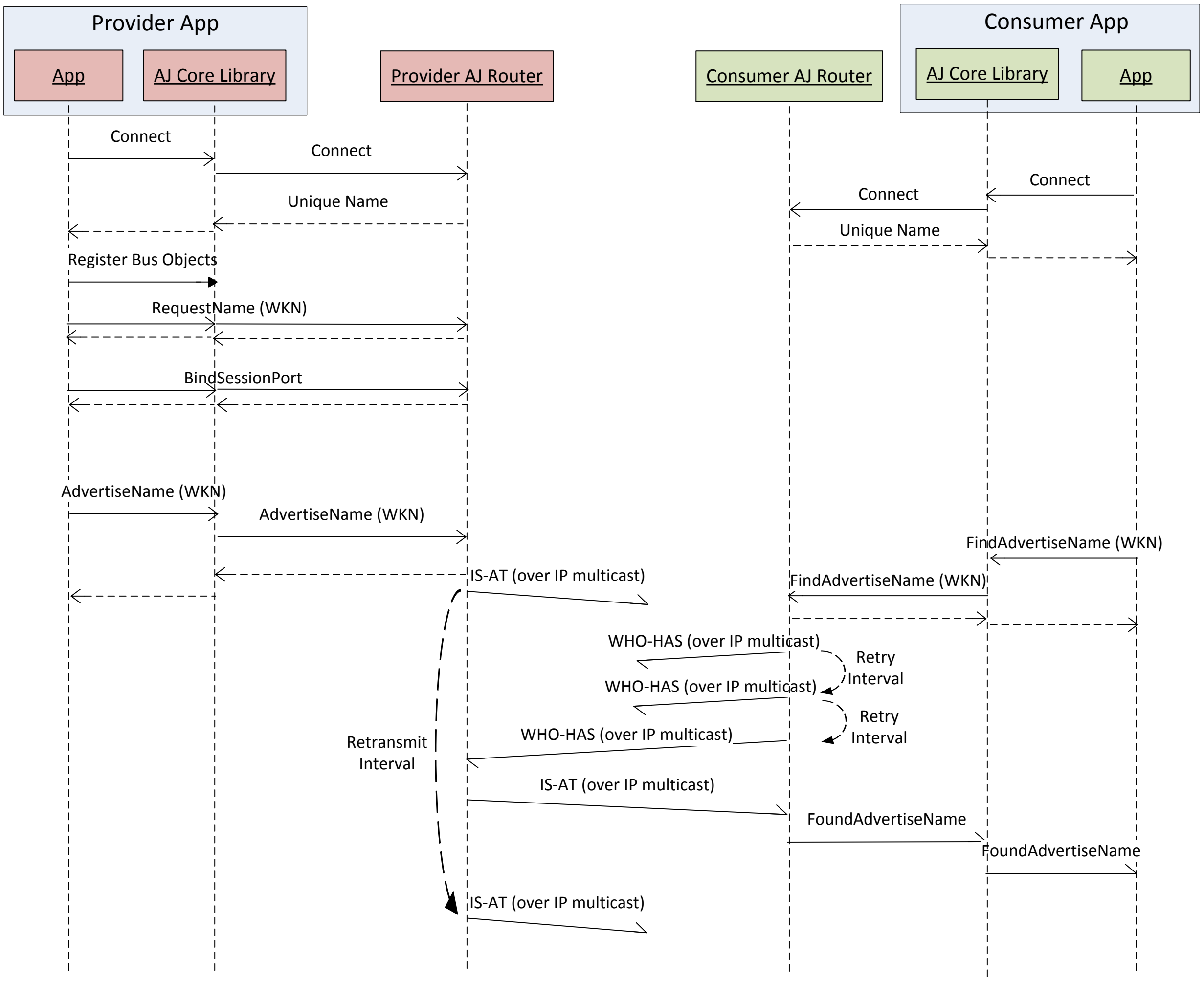


Figure 4-4. Discovery over unreliable network

4.2.4.3 Discovery when IP connectivity is established late

Figure 4-5 shows the message sequence for the discovery scenario when the consumer device gets connected to the Access Point (AP) in the AllJoyn proximal network late, after it has completed transmission of set of WHO-HAS messages. This can happen when the consumer device just joins a new AllJoyn proximal network. The subsequent IS-AT message is received by the consumer AllJoyn router and results in FoundAdvertiseName for the requested well-known name.

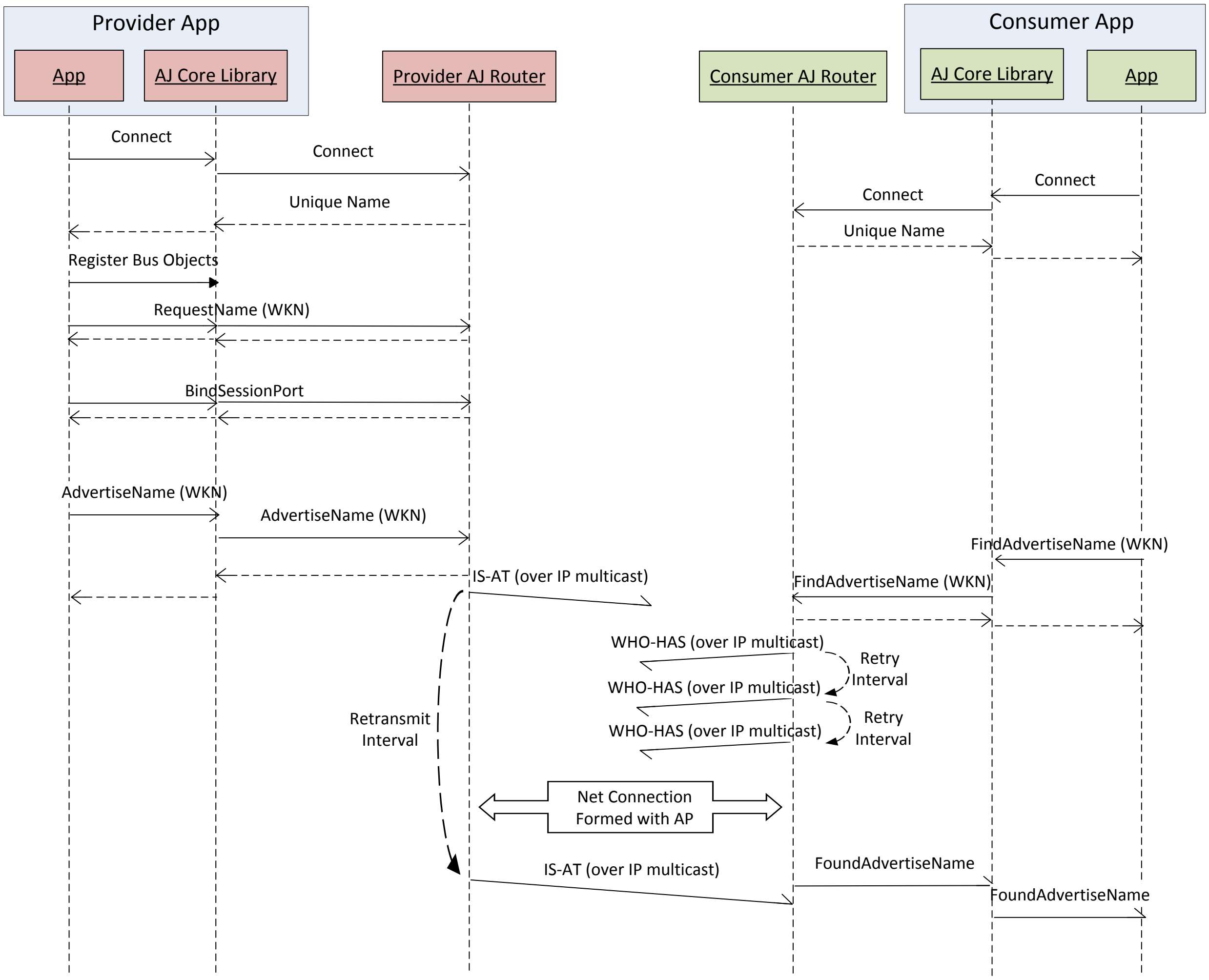


Figure 4-5. Discovery when IP connectivity is established late

4.2.4.4 WKN lost due to loss of IP connectivity

Figure 4-6 shows the message sequence for the scenario when the discovered well-known name gets lost due to the consumer losing IP connectivity with the AllJoyn proximal network. This can happen when a consumer device leaves the AllJoyn proximal network.

If the consumer's AllJoyn router does not receive any IS-AT messages for a given well-known name for the Adv_Validity_period time duration, it declares that well-known name to be lost and initiates a LostAdvertiseName for that well-known name.

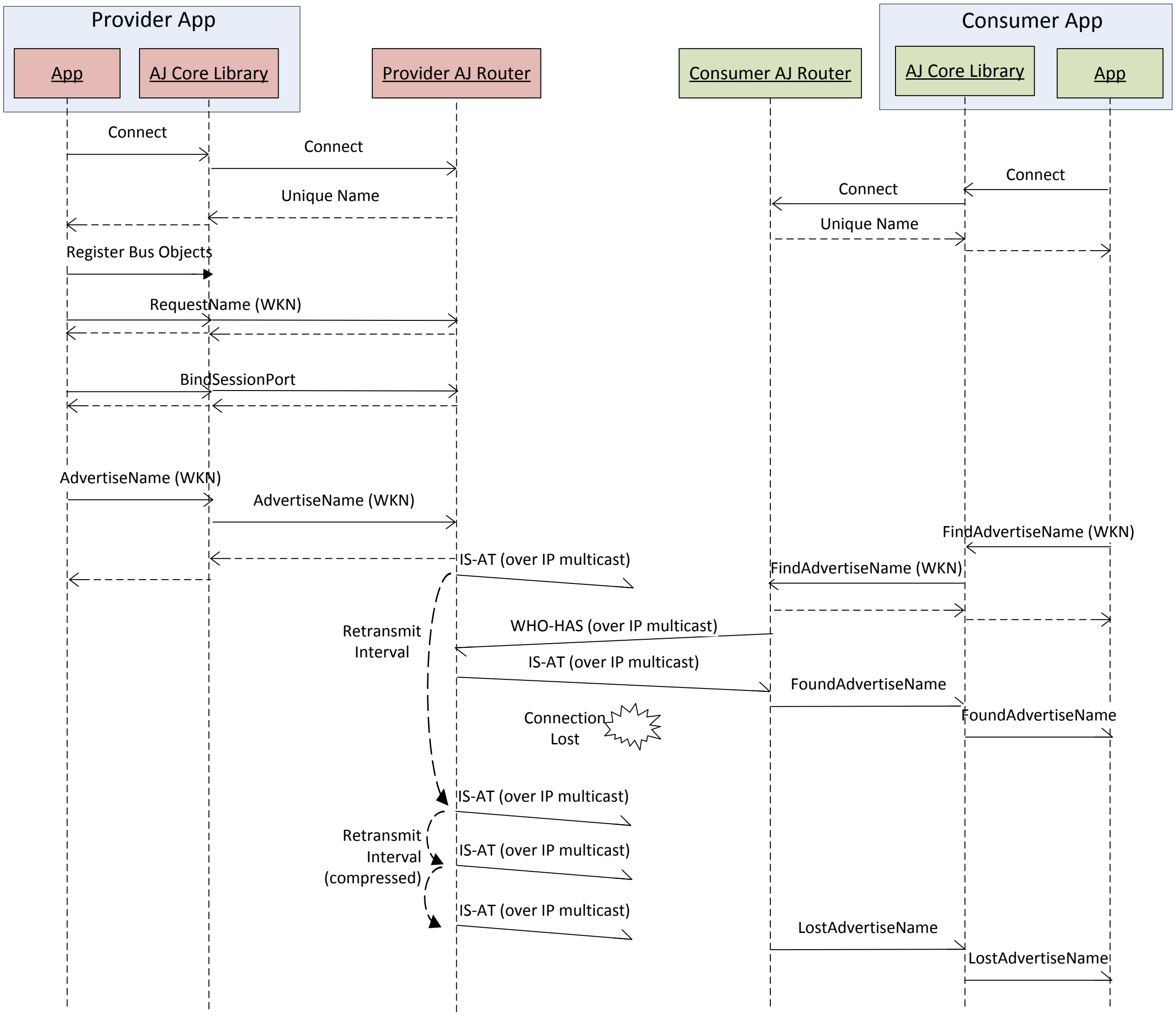


Figure 4-6. Well-known name lost due to loss of IP connectivity

4.2.4.5 Provider cancels well-known name advertisement

Figure 4-7 shows the message sequence for the scenario when a provider application cancels the advertisement for a previously advertised well-known name.

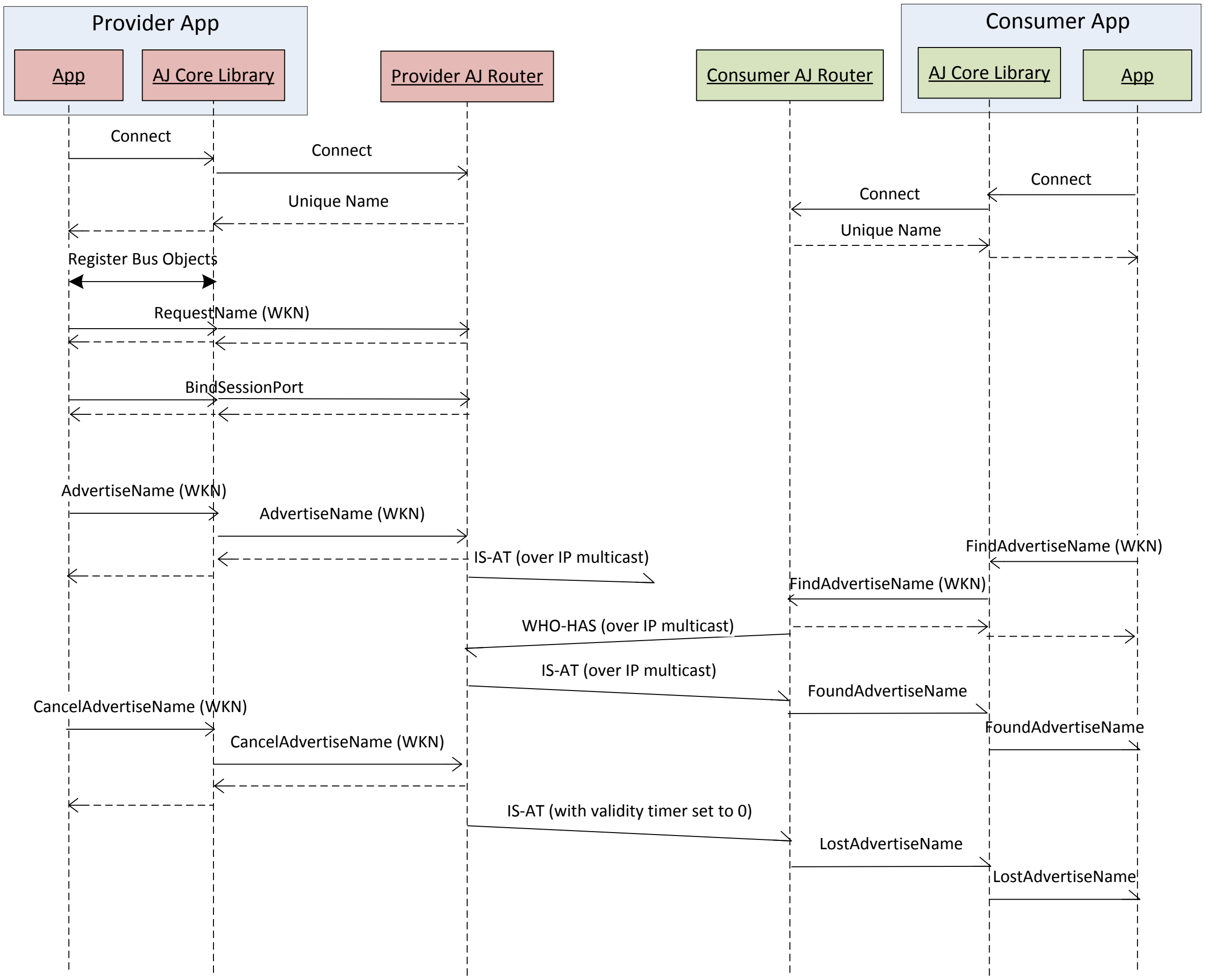


Figure 4-7. Provider cancels well-known name advertisement

4.2.4.6 Consumer cancels discovery for well-known name

Figure 4-8 shows the message sequence for the scenario when a consumer application cancels discovery for a well-known name.

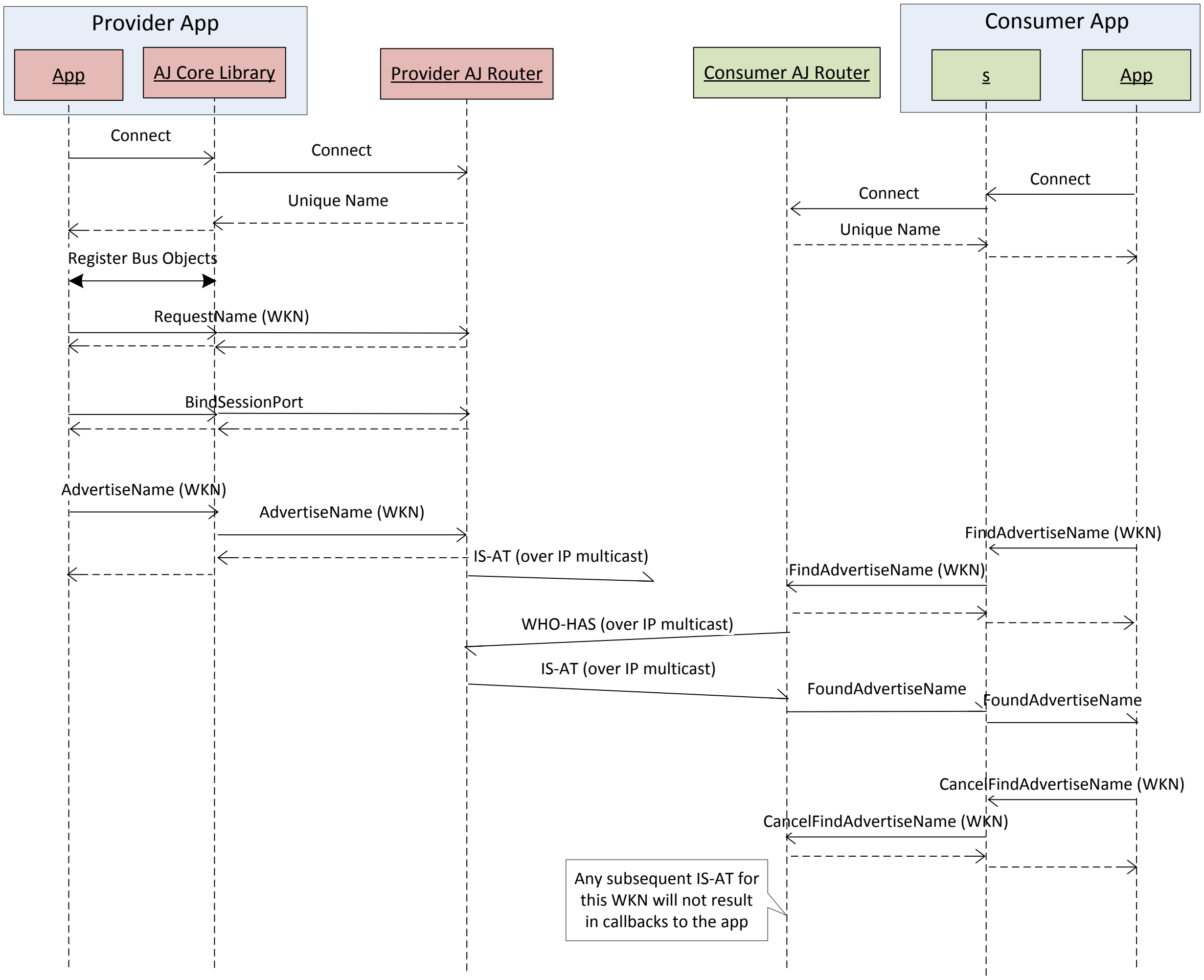


Figure 4-8. Consumer cancels discovery for well-known name

4.2.5 Message structure

As described above, the Name Service supports IS-AT and WHO-HAS messages. These messages get embedded in a higher-level Name Service message which provides the flexibility to include both IS-AT and WHO-HAS messages in the same Name Service message. This can be useful if an AllJoyn application acts as both provider (advertising a well-known name) and a consumer (looking to discover a well-known name).

Figure 4-9 shows the Name Service message structure. Table 4-2 defines the message structure fields.

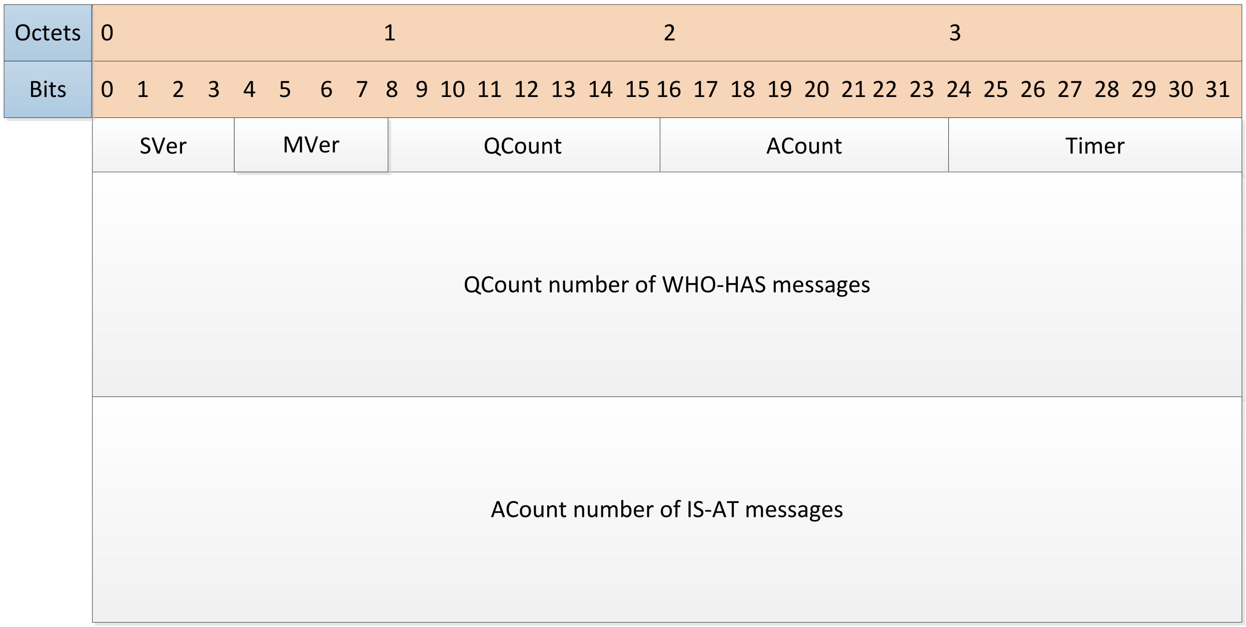


Figure 4-9. Name service message structure

Table 4-2. Name Service message structure fields

Field	Description
Sver	Version of the latest implemented the AllJoyn discovery protocol for the sender.
MVer	Version of Name Service message.
QCount	Number of WHO-HAS question messages that follow the header.
ACount	Number of IS-AT answer messages that follow the header.
Timer	Count (in seconds) for which included IS-AT answer should be considered valid. This field should be set based on the following: ■ Adv_Validity_Period for a well-known name advertisement that is valid for a default period of time. ■ Adv_Infinite_Validity_Value for a well-known name advertisement that is valid “forever”, or at least until withdrawn. A zero in this field means that the sending AllJoyn router is withdrawing the advertisements.

4.2.5.1 IS-AT message

Figure 4-10 shows version 1 of the IS-AT message. Table 4-4 defines the IS-AT message fields.

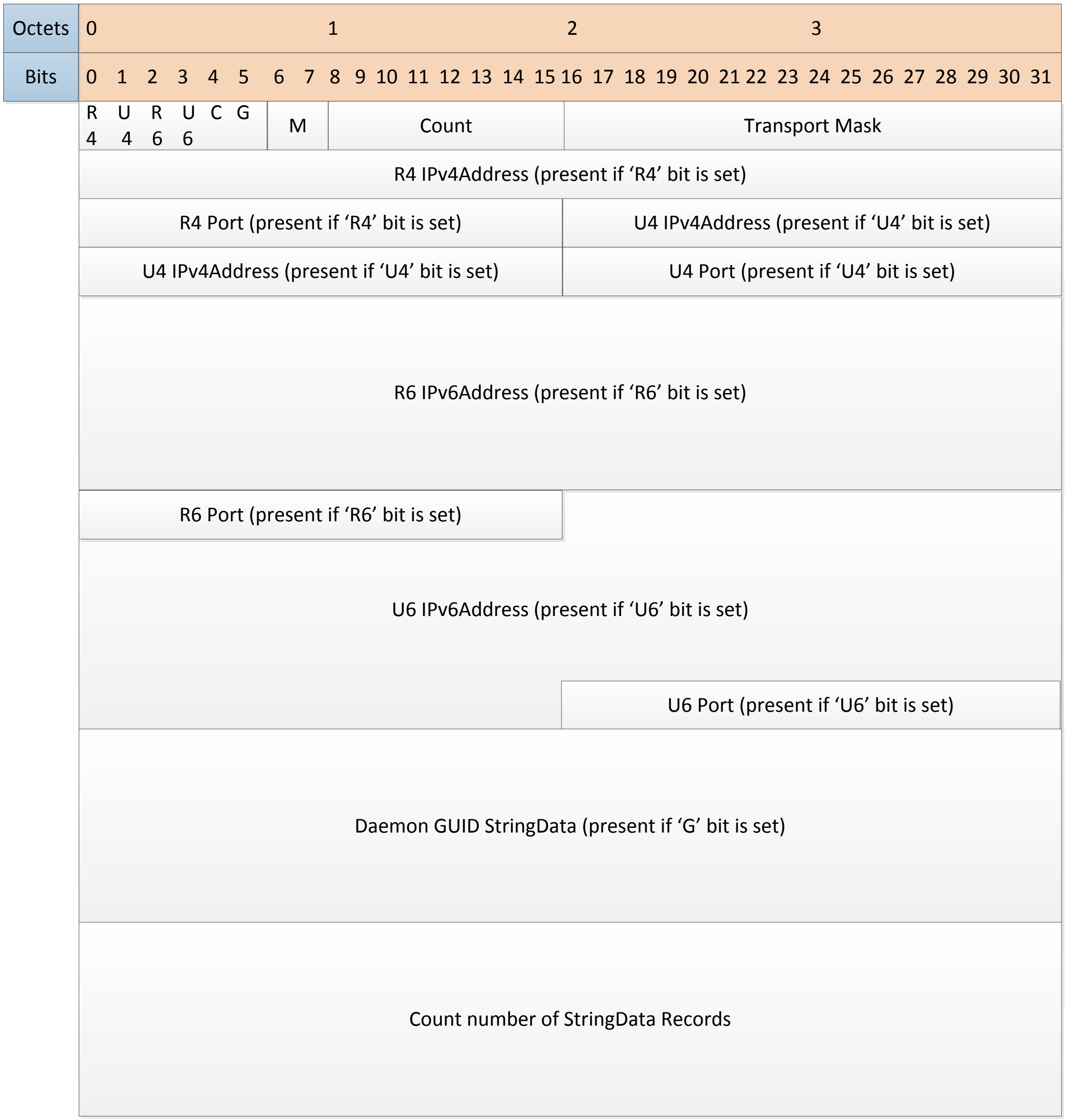

Octets
0
1
2
3
Bits
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
R4
U4
R6
U6
C
G
M
Count
Transport Mask
R4 IPv4Address (present if 'R4' bit is set)
R4 Port (present if 'R4' bit is set)
U4 IPv4Address (present if 'U4' bit is set)
U4 IPv4Address (present if 'U4' bit is set)
U4 Port (present if 'U4' bit is set)
R6 IPv6Address (present if 'R6' bit is set)
R6 Port (present if 'R6' bit is set)
U6 IPv6Address (present if 'U6' bit is set)
U6 Port (present if 'U6' bit is set)
Daemon GUID StringData (present if 'G' bit is set)
Count number of StringData Records

Figure 4-10. IS-AT Message format (version 1)

Table 4-3. IS-AT message format (version 1) fields

Field	Description
R4 Bit	If set to '1', the R4 bit indicates that the IPv4 endpoint (IP address and port) of a reliable transport (TCP) is present.
U4 Bit	If set to '1', the U4 bit indicates that the IPv4 endpoint (IP address and port) of an unreliable transport (UDP) is present.
R6 Bit	If set to '1', the R6 bit indicates that the IPv6 endpoint (IP address and port) of a reliable transport (TCP) is present.

Field	Description
U6 Bit	If set to '1', the U6 bit indicates that the IPv6 endpoint (IP address and port) of an unreliable transport (UDP) is present.
C Bit	If set to '1', the C bit indicates that the list of StringData records is a complete list of all well-known names exported by the responding AllJoyn router.
G Bit	If set to ‘1’, the G bit indicates that a variable length daemon GUID string is present.
M	Message type of the IS-AT message. Defined to be '01' (1) for IS-AT.
Count	Number of StringData items that are included in the IS-AT message.
TransportMask	Bit mask of transport identifiers that indicates which AllJoyn transport is making the advertisement.
StringData	Describes a single AllJoyn well-known name being advertised.

4.2.5.2 WHO-HAS message

Figure 4-11 shows version 1 of the WHO-HAS message. Table 4-4 defines the WHO-HAS message fields.

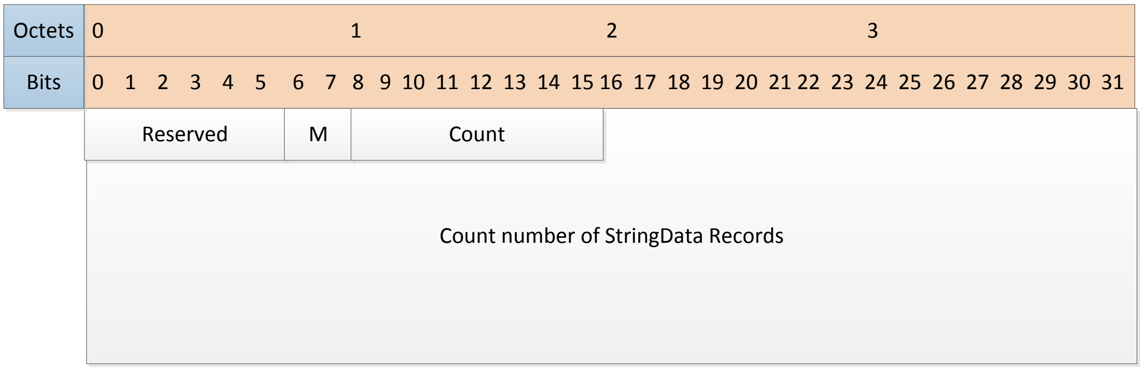


Figure 4-11. WHO-HAS Message format (version 1)

Table 4-4. WHO-HAS message format (version 1) fields

Field	Description
Reserved	Reserved bits
M	Message type of the WHO-HAS message. Defined to be '10' (2) for WHO-HAS.
Count	Number of StringData items that are included in the WHO-HAS message.
StringData	Describes a single AllJoyn well-known name that the consumer AllJoyn router is interested in.

4.3 Legacy announcement-based discovery

This section captures design for the legacy announcement-based discovery supported prior to the 14.06 release.

In the announcement-based discovery, the provider device announces the set of AllJoyn interfaces supported via an announcement broadcast signal. The consumer device interested in making use of the AllJoyn services opts to receive these broadcast announcement messages from providers to discover the interfaces for the supported AllJoyn services.

The Announcement message is generated by the About feature and is delivered as an AllJoyn sessionless signal using the sessionless signal mechanism provided by the AllJoyn router (detailed Chapter 7). The sessionless signal module makes use of the AllJoyn name service messages (IS-AT and WHO-HAS) to notify the consumer of new signals using a specially formatted well-known name for the sessionless signal. Once the consumer AllJoyn router discovers the sessionless signal's well-known name, it connects back to the provider over an AllJoyn session to fetch the service announcement message from the provider device.

Figure 4-12 captures the high-level architecture for the announcement-based discovery process.

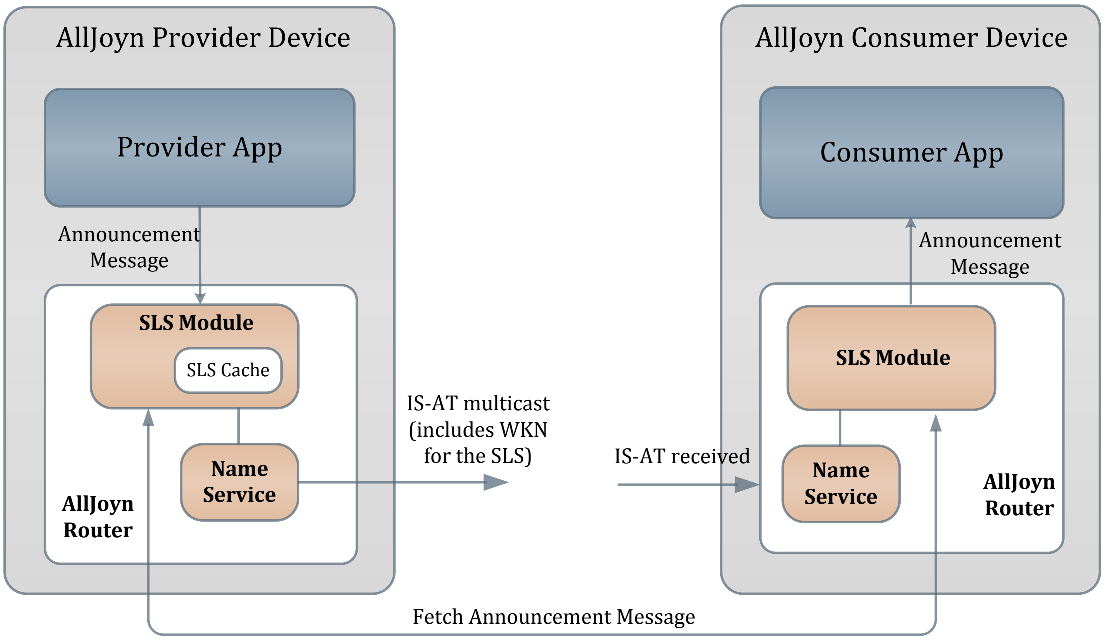


Figure 4-12. Announcement-based service discovery architecture

The Announcement message is sent as a sessionless signal from the provider app to the AllJoyn router, and gets cached in the sessionless signal cache. The sessionless signal module generates a specially formatted well-known name for the sessionless signal as shown below (see details in Chapter 7):

SLS WKN format: org.alljoyn.sl.x<GUID>.x<change_id>

The sessionless signal module interacts with the Name Service to send an IS-AT message for that well-known name. The AllJoyn router on the consumer side is looking to discover this well-known name. Upon receiving the IS-AT message, the sessionless signal module on the consumer side connects back to the sessionless signal module on the provider via an AllJoyn session and fetches the Announcement message which then gets delivered to the consumer app.

4.3.1 Message sequence

Figure 4-13 shows the message sequence for the announcement-based discovery.

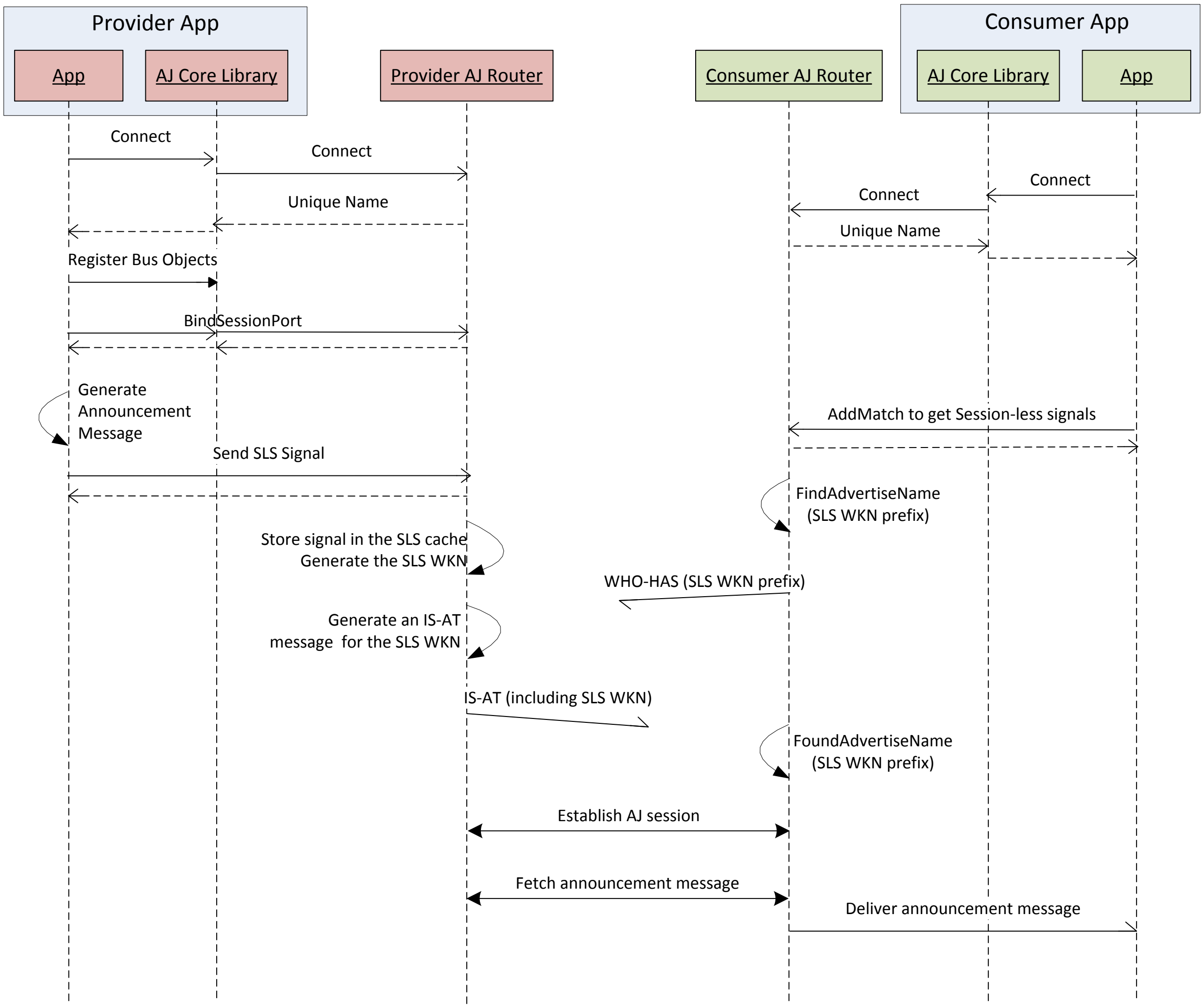

Provider App
App
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer App
AJ Core Library
App
Connect
Connect
Unique Name
Connect
Connect
Unique Name
Register Bus Objects
BindSessionPort
Generate Announcement Message
Send SLS Signal
AddMatch to get Session-less signals
FindAdvertiseName (SLS WKN prefix)
Store signal in the SLS cache
Generate the SLS WKN
WHO-HAS (SLS WKN prefix)
Generate an IS-AT message for the SLS WKN
IS-AT (including SLS WKN)
FoundAdvertiseName (SLS WKN prefix)
Establish AJ session
Fetch announcement message
Deliver announcement message

Figure 4-13. Announcement-based service discovery message sequence

4.3.2 Announcement message

The Announcement message provides a list of object paths for objects implemented by the AllJoyn application and AllJoyn interfaces supported by each of those objects. The AllJoyn application controls which objects get announced in the Announcement message.

The Announcement message also contains additional About fields describing information about the application and the device. Refer to [3] for Announcement message details.

4.4 Legacy AllJoyn discovery configuration parameters

Table 4-5 captures configuration parameter for legacy AllJoyn discovery.

NOTE

Implementation may use different names for these parameters.

Table 4-5. AllJoyn discovery configuration parameters

Parameter	Default value	Range	Description
Adv_Validity_Period	120 seconds	TBD	Validity period used for IS-AT advertisements.
Adv_Infinite_Validity_Value	255	TBD	Time value for indicating that an advertisement is valid forever.
Adv_Msg_Retransmit_Interval	40 seconds	TBD	Interval in seconds for sending out IS-AT messages.
Disc_Msg_Number_Of_Retries	2	TBD	Number of times the WHO-HAS message is sent after the first transmission.
Disc_Msg_Retry_Interval	5 seconds	TBD	Interval in seconds between retries of the WHO-HAS message.

4.5 Next-generation name service

The Next-Generation Name Service (NGNS) is implemented in the 14.06 release and offers considerable performance enhancements for discovery and presence features offered by the AllJoyn platform, detailed in the subsequent relevant sections.

Figure 4-14 shows the high-level architecture for NGNS.

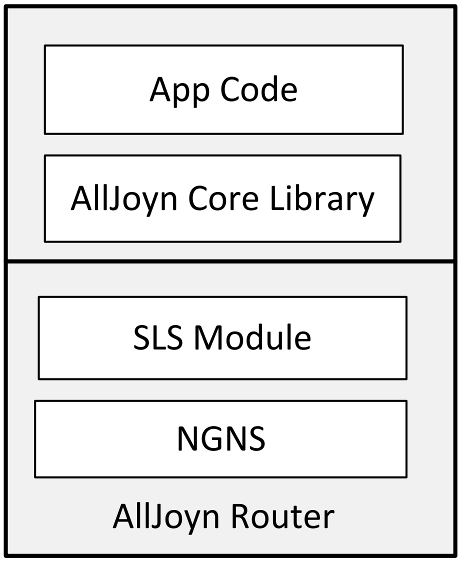


Figure 4-14. NGNS high-level architecture

The architecture shows main logical components related to NGNS. The enhanced discovery and presence functionality are exposed via new APIs as part of the AllJoyn core library. The About functionality is included in the AllJoyn core library, and enables an AllJoyn app to send Announcement sessionless signals. The sessionless signal module caches the Announcement signal. The NGNS module uses information in the Announcement signal to answer interface-based discovery queries received from consumer apps.

4.5.1 Discovery

The AllJoyn framework offers name-based discovery or announcement-based discovery as mentioned earlier in this chapter. NGNS supports the following discovery mechanisms:

- NGNS supports name-based discovery. Although there is no change at the API level, the discovery utilizes DNS service discovery framework over mDNS. NGNS sends out legacy (pre-14.06 release) discovery messages as per the configuration setting in the AllJoyn router config file for compatibility.
- NGNS supports a more efficient announcement-based discovery process by allowing a consumer application to query for a set of AllJoyn interfaces. Prior to the 14.06 release, the consumer application had to create match rules to receive all Announce signals (transmitted as sessionless signals), and parse through the set of AllJoyn interfaces that the provider application is announcing prior to making a determination if any interfaces of interest are provided. While this mechanism is more powerful than the well-known name-based mechanism, it was not efficient. The NGNS feature allows a consumer application to query for the set of AllJoyn interfaces, and only the provider applications that make use of those interfaces answer the query.

4.5.2 Presence detection

Prior to the 14.06 release, presence (or absence) detection was based on three successive IS-AT messages missing for a given name (well-known or unique name) by the consumer application. The time taken for this detection was deterministic (3*40 sec = 120 sec).

The use of NGNS in the 14.06 release introduces an efficient consumer application-driven presence detection that makes use of unicast messaging. Once a name has been discovered, the consumer application can invoke the new Presence API and determine the presence state. Since each application has its own logic regarding times and events triggering presence detection, NGNS provides the API and leaves the triggering logic for the application to drive.

4.5.3 NGNS design aspects

The following sections detail the design aspects of the NGNS feature.

4.5.3.1 Usage of mDNS

The 14.02 discovery protocol is based on AllJoyn-specific UDP messages over the AllJoyn-assigned multicast IP address. This design can limit discoverability (IP routers can block AllJoyn-assigned multicast IP address and/or port numbers) in the field. To address this issue, the 14.06 discovery protocol is based on multicast DNS (mDNS) that uses IANA-assigned multicast IP address and port numbers.

Table 4-6 lists the multicast addresses and port number used by NGNS.

Table 4-6. Multicast IP address and port numbers used by NGNS

Address	Value
IPv4 Multicast group address	224.0.0.251
IPv6 Multicast group address	FF02::FB
Multicast port number	5353

Furthermore, mDNS already supports the following features that are utilized by the AllJoyn discovery protocol:

- Solicit unicast responses
- Send query message using unicast
- Send unsolicited responses from the responder

This constitutes version 2 of the discovery protocol. The version number is set in the pv field of the “sender-info” TXT record in the additional section of the mDNS query and response.

NOTE

The 14.02 Name Service implementation uses version 0 and 1 of the discovery protocol.

4.5.3.2 Usage of DNS-SD

The 14.06 discovery design is based on [9].

A client discovers the list of available instances of a given service name (as registered with IANA, e.g., alljoyn is a registered service name) using a query for a DNSPTR record with a name of the form:

"<Service>.<Domain>" [5].

The result of this PTR lookup for the name "<Service>.<Domain>" is a set of zero or more PTR records giving Service Instance Names of the form:

Service Instance Name = <Instance>.<Service>.<Domain>

In addition to that service instance, the DNS-SD responder sends DNS SRV [10] and DNS TXT [5] record. The SRV and TXT records have a name of the form:

"<Instance>.<Service>.<Domain>"

The SRV record gives the target host and port where the service instance can be reached. The DNS TXT record of the same name gives additional information about this instance, in a structured form using key/value pairs.

In addition to the service discovery framework specified in [9], the NGNS discovery protocol sends DNS TXT records in the Additional section of the DNS-SD query to optimize the discovery scope without requiring further negotiation by establishing an AllJoyn session with the provider application. The same feature is utilized in other use cases, such as sending sender-information or presence-related information. The DNS-SD message format is described in detail in section 4.7.

4.5.3.3 Design considerations for Wi-Fi

It is well known that the multicast success rate over Wi-Fi is not optimal and in some cases it is substantially degraded. As per the Wi-Fi specification, each station is allowed to go into sleep state and wake up periodically. The wake-up interval is provisioned at the device and is supposed to be factor of the time interval used by the AP to schedule the multicast traffic.

The AP buffers the incoming multicast data and schedules it based on the time interval determined by the DTIM (Delivery Traffic Indication Message) interval. In reality, it has been observed that the wake-up interval is set to multiples of the DTIM value (typically 1, 3, or 10). This implies that the device might miss the multicast data. Since this is a realistic scenario, the AllJoyn discovery protocol was designed to handle this scenario in a robust fashion.

Specifically, the design principles captured in sections below are adopted.

4.5.3.3.1 Transmission schedule

The multicast schedule was designed to support devices that wake up to process multicast packets in the multiple of DTIM interval. Although the schedule backs off exponentially, each multicast message is repeated twice to improve reliability of multicast messages with devices that wake up every third DTIM interval (a very typical case).

The schedule sends query messages at the following times: 0, 1, 3, 9, and 27 seconds. At each transmit time trigger, a total of three messages (original plus two repeats 100 msec apart) are sent. This is referred to as burst in the call flows. As far as the message recipient is concerned, a response to the first successfully received message in a burst is sent and all subsequent messages that are part of the same burst are ignored.

4.5.3.3.2 Minimize multicast and maximize unicast transmissions

Another design aspect is to send multicast messages to initiate queries but rely on unicast responses for replies and presence detection. The DNS allows unicast responses to be solicited in the mDNS query, and the discovery protocol utilizes that feature. This is indicated in the top bit of the qclass field of the DNS message header [8].

4.5.3.4 Discovery and Presence API snapshot

Table 4-7 lists the Discovery and Presence APIs offered by the AllJoyn system and maps them to discovery scenarios. The main paradigm is that discovery and presence are driven by the consumer application.

Table 4-7. Discovery and Presence APIs related to discovery scenarios

Discovery scenario	API
Consumer application Name query	FindAdvertisedName()
Consumer application gets notified about discovery or loss of an advertised name	■ FoundAdvertisedName() ■ LostAdvertisedName()
Consumer application cancelling the Name query	CancelFindAdvertisedName()
Provider application advertising a name	AdvertiseName()
Providing application canceling advertising a name	CancelAdvertiseName()
Provider application sending an Announcement message	Announce()
Consumer application queries for set of AllJoyn interfaces	RegisterAnnounceHandler()
Consumer application cancels a query for set of AllJoyn interfaces	UnregisterAnnounceHandler()
Consumer application queries for presence	Ping()

4.5.3.4.1 Discovery APIs that trigger DNS-SD multicast messages

Some of the discovery scenarios trigger multicast messaging. The APIs that trigger multicast messaging are: FindAdvertisedName(), CancelAdvertiseName(), AdvertiseName(), Announce() and RegisterAnnounceHandler().

Some key aspects of the multicast transmission are listed below:

- DNS-SD query issued over mDNS multicast address
- Transmission schedule as per section 4.5.3.3.1
- Name Service messages sent depending on the LegacyNS flag setting in the Router config file
- Legacy Name Service WHO-HAS and mDNS messages follow the same transmission schedule.

4.5.3.5 Backward compatibility

NGNS is designed to meet the following backward compatibility requirements:

- Support all existing 14.02 APIs
- Support all legacy (version 0 and 1) NS discovery packet formats
- Send equivalent 14.02 discovery message over the wire whenever the corresponding DNS-SD message is being sent (provided the LegacyNS flag is set to true)
- Replies with 14.02 response message upon receipt of a 14.02 query message provided the supported version (SVer) field indicates that querier doesn’t support NGNS. If the querier supports NGNS as indicated by the supported version, then NGNS waita for the DNS-SD messages to arrive.

The AllJoyn router configuration file adds a LegacyNS flag to enable legacy discovery behavior. By default, the legacy behavior is enabled.

4.6 NGNS message sequences

This section captures message sequences for NGNS.

4.6.1 Name-based discovery

This section captures messages sequences for NGNS name-based discovery scenarios.

4.6.1.1 NGNS consumer app with NGNS provider app

In this scenario, the consumer application's AllJoyn router has disabled the legacy behavior, i.e., no Name Service messages are being sent by the consumer application.

This message sequence assumes that the provider application is already on the AllJoyn network.

The main steps for the message sequence are described below.

1. The message flow is initiated by the consumer application invoking FindAdvertisedName().
2. NGNS sends DNS-SD based query messages over mDNS.
3. Any provider application that matches the name being searched responds via the DNS-SD response message over unicast to the consumer application.

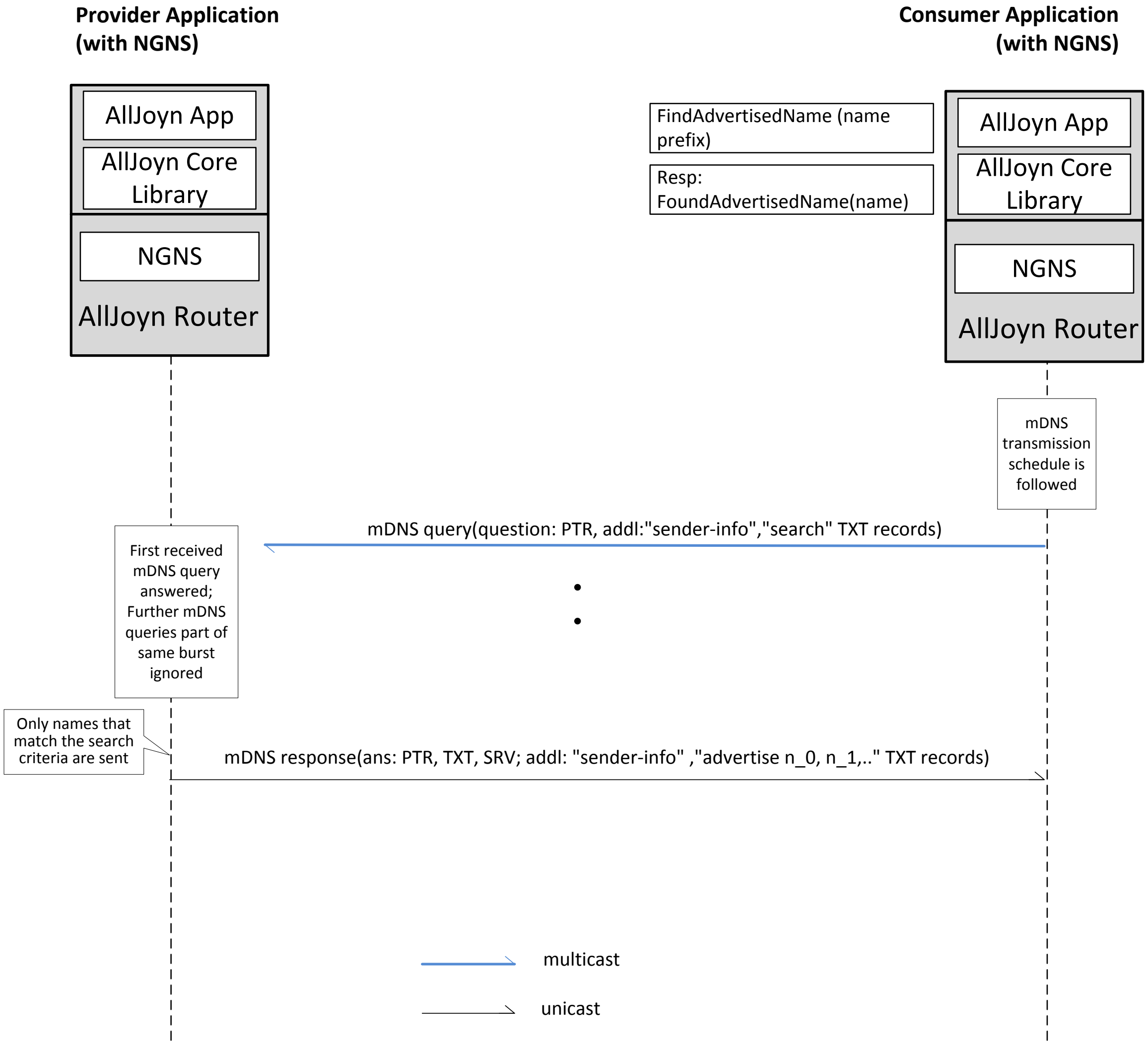

Provider Application (with NGNS)
Consumer Application (with NGNS)
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
FindAdvertisedName (name prefix)
Resp: FoundAdvertisedName(name)
mDNS transmission schedule is followed
mDNS query(question: PTR, addl:"sender-info","search" TXT records)
First received mDNS query answered; Further mDNS queries part of same burst ignored
Only names that match the search criteria are sent
mDNS response(ans: PTR, TXT, SRV; addl: "sender-info" ,"advertise n_0, n_1,.." TXT records)
multicast
unicast

Figure 4-15. NGNS name-based discovery between consumer app and provider app

4.6.1.2 NGNS consumer app with NGNS and Name Service provider apps

This message sequence assumes the following:

- The consumer application's AllJoyn router has enabled the legacy Name Service behavior.
- The provider applications are already on the AllJoyn network.

The main steps for the message sequence are described below.

1. The message flow is initiated by the consumer application invoking FindAdvertisedName().
2. The NGNS sends DNS-SD based query messages over mDNS as well as the legacy WHO-HAS message.

3. Any AllJoyn provider application that matches the name being searched responds via a DNS-SD response message over unicast
4. Any legacy (14.02) provider application also responds via an IS-AT message if there is a match for the name being discovered in the WHO-HAS message.

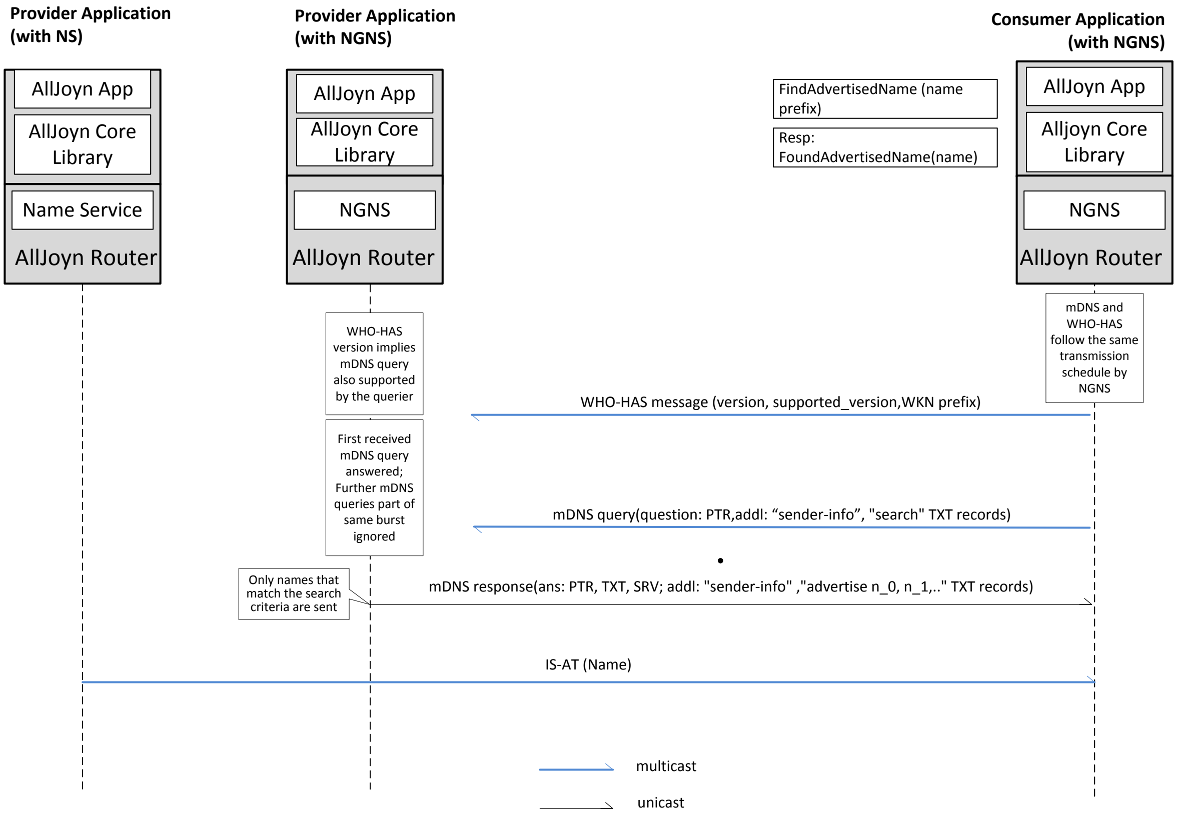

Provider Application (with NS)
Provider Application (with NGNS)
Consumer Application (with NGNS)
AllJoyn App
AllJoyn Core Library
Name Service
AllJoyn Router
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
FindAdvertisedName (name prefix)
Resp: FoundAdvertisedName(name)
AllJoyn App
Alljoyn Core Library
NGNS
AllJoyn Router
mDNS and WHO-HAS follow the same transmission schedule by NGNS
WHO-HAS version implies mDNS query also supported by the querier
WHO-HAS message (version, supported_version,WKN prefix)
First received mDNS query answered; Further mDNS queries part of same burst ignored
mDNS query(question: PTR,addl: “sender-info”, "search" TXT records)
Only names that match the search criteria are sent
mDNS response(ans: PTR, TXT, SRV; addl: "sender-info" ,"advertise n_0, n_1,.." TXT records)
IS-AT (Name)
multicast
unicast

Figure 4-16. NGNS name-based discovery (NGNS and Name Service provider apps)

4.6.1.3 FindAdvertisedName pending; provider apps arrive later

This message sequence assumes the following:

- The consumer application’s AllJoyn router has enabled the legacy Name Service behavior.
- The provider applications are not on the AllJoyn network at the time of initial query.

The main steps for the message sequence are described below.

1. The message flow is initiated by the consumer application invoking FindAdvertisedName().
2. The NGNS sends DNS-SD based query messages over mDNS, as well as legacy Name Service messages.

3. The query schedule for mDNS messages and WHO-HAS messages expires
4. Upon joining the AllJoyn network, the NGNS provider app sends unsolicited DNS-SD response messages and advertises names via IS-AT messages.
5. Upon joining the AllJoyn network, the Name Service provider application sends IS-AT messages.
6. The consumer AllJoyn router performs the following tasks:
   a. It consumes the Name Service and NGNS messages.
   b. It filters the names being advertised.
   c. It sends FoundAdvertisedName() only if there is a match.

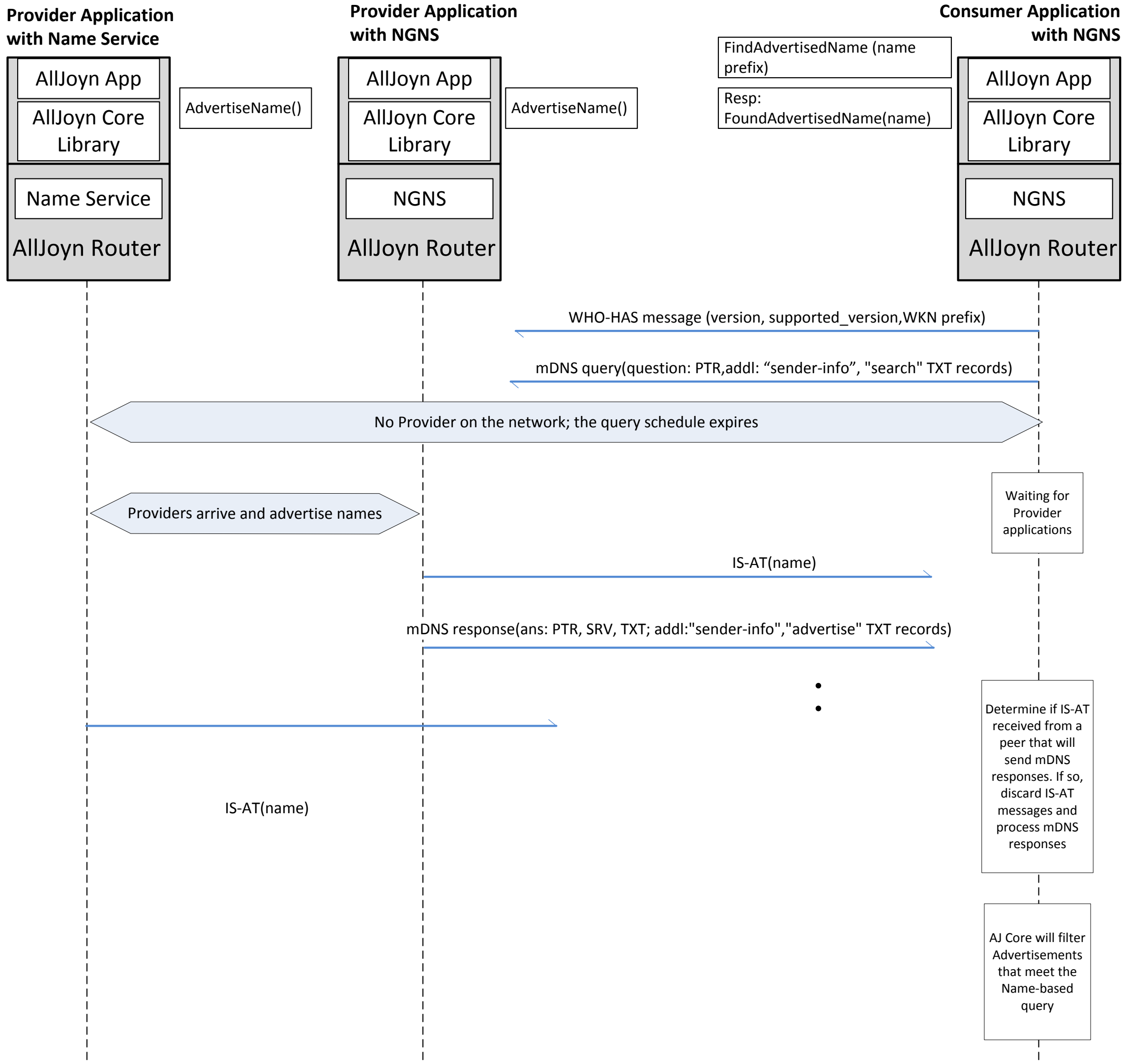

Provider Application with Name Service
AllJoyn App
AllJoyn Core Library
Name Service
AllJoyn Router
AdvertiseName()
Provider Application with NGNS
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
AdvertiseName()
FindAdvertisedName (name prefix)
Resp: FoundAdvertisedName(name)
Consumer Application with NGNS
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
WHO-HAS message (version, supported_version,WKN prefix)
mDNS query(question: PTR,addl: "sender-info", "search" TXT records)
No Provider on the network; the query schedule expires
Waiting for Provider applications
Providers arrive and advertise names
IS-AT(name)
mDNS response(ans: PTR, SRV, TXT; addl:"sender-info","advertise" TXT records)
Determine if IS-AT received from a peer that will send mDNS responses. If so, discard IS-AT messages and process mDNS responses
IS-AT(name)
AJ Core will filter Advertisements that meet the Name-based query

Figure 4-17. FindAdvertisedName API called; provider arrives later

4.6.2 Interface names discovery

4.6.2.1 NGNS consumer app and NGNS provider app

This message sequence assumes the following:

- The provider application is already on the AllJoyn network.
- The legacy Name Service behavior is turned off on the consumer AllJoyn router.

The main steps for the message sequence are described below.

1. The message flow is initiated by the consumer application registering the announce handler (by calling RegisterAnnounceHandler) and providing a set of AllJoyn interfaces. This triggers the discovery for provider applications that implement those interfaces.
2. The NGNS sends DNS-SD based query messages over mDNS and populates the search TXT record in the Additional section based on the AllJoyn interfaces being discovered.
3. Any AllJoyn provider application that provides the AllJoyn interfaces being discovered sends the DNS-SD response message and includes the sessionless signal well-known name corresponding to the About Announce signal.

   Note that this message is sent over unicast
4. The consumer application immediately initiates a sessionless signal fetch to retrieve the Announce signal.

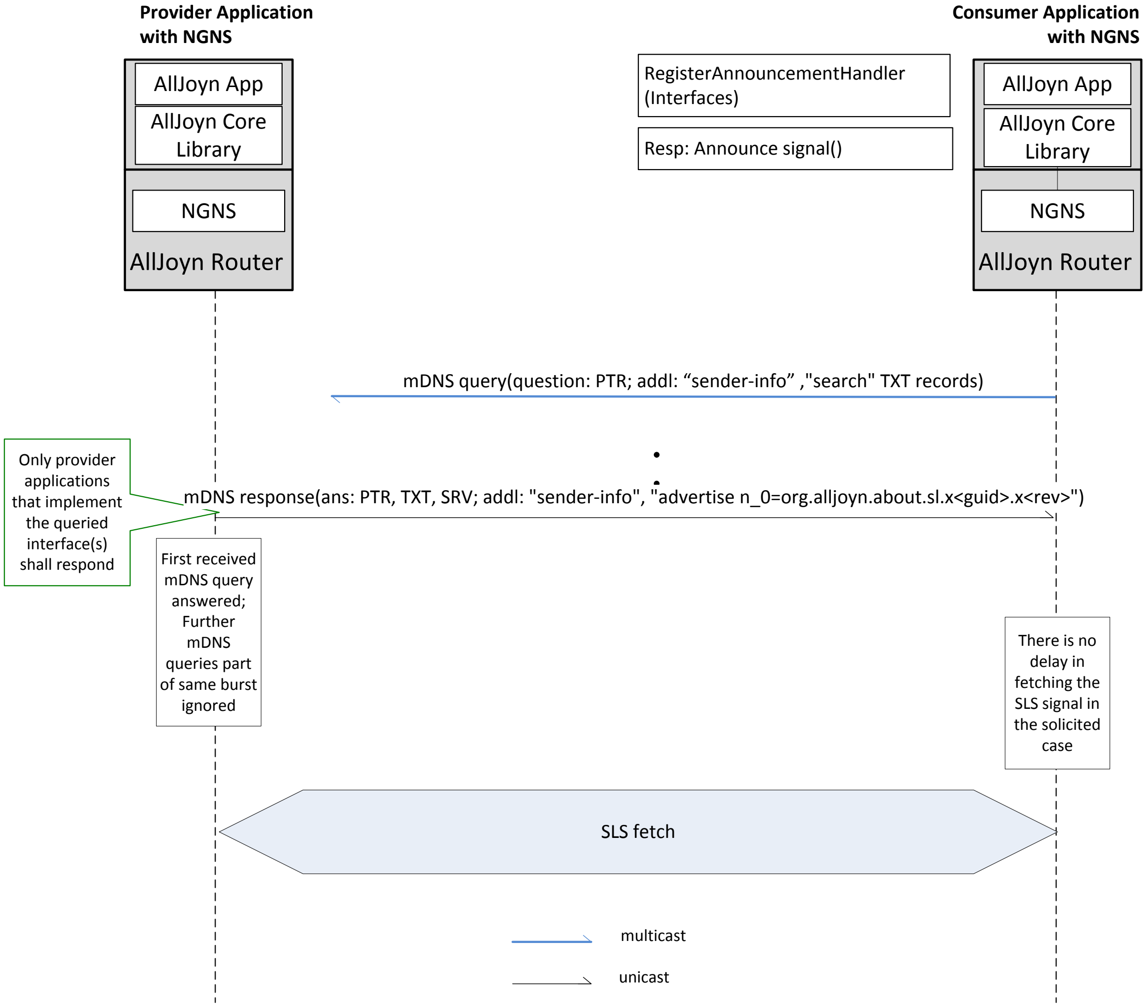

Provider Application with NGNS
Consumer Application with NGNS
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
RegisterAnnouncementHandler (Interfaces)
Resp: Announce signal()
mDNS query(question: PTR; addl: “sender-info” ,"search" TXT records)
Only provider applications that implement the queried interface(s) shall respond
mDNS response(ans: PTR, TXT, SRV; addl: "sender-info", "advertise n_0=org.alljoyn.about.sl.x<guid>.x<rev>")
First received mDNS query answered; Further mDNS queries part of same burst ignored
There is no delay in fetching the SLS signal in the solicited case
SLS fetch
multicast
unicast

Figure 4-18. AllJoyn interface query (NGNS consumer app and NGNS provider app)

4.6.2.2 NGNS consumer app; NGNS and Name Service provider app

This message sequence is an extension of the call flow in Figure 4-18 with the legacy Name Service behavior being enabled.

Although the interface-based query is a 14.06 feature, it has been designed such that legacy Name Service provider applications can participate in the discovery process. This is enabled by sending WHO-HAS message with WKN=org.alljoyn.sl.

This message sequence assumes that the provider application is already on the AllJoyn network.

The main steps for the message sequence are described below:

1. The message flow is initiated by the consumer application registering the announce handler (by calling RegisterAnnounceHandler) and providing a set of AllJoyn interfaces for discovery.

2. The NGNS sends DNS-SD based query messages over mDNS, and populates the search TXT record in the Additional section based on the AllJoyn interfaces being discovered.
3. The NGNS sends WHO-HAS discovery messages with WKN=org.alljoyn.sl.
4. Any NGNS provider application that provides the AllJoyn interfaces being discovered sends the DNS-SD response message and includes the sessionless signal well-known name corresponding to the About Announce signal.

   Note that this message is sent over unicast.
5. The consumer application immediately initiates a sessionless signal fetch to retrieve the Announce signal.
6. Any legacy provider application sends an IS-AT message with the sessionless signal well-known name if there are any sessionless signals in the sessionless signal cache.
7. The consumer application immediately initiates sessionless signal fetch and filters the Announce signals that provide the AllJoyn interfaces being discovered.

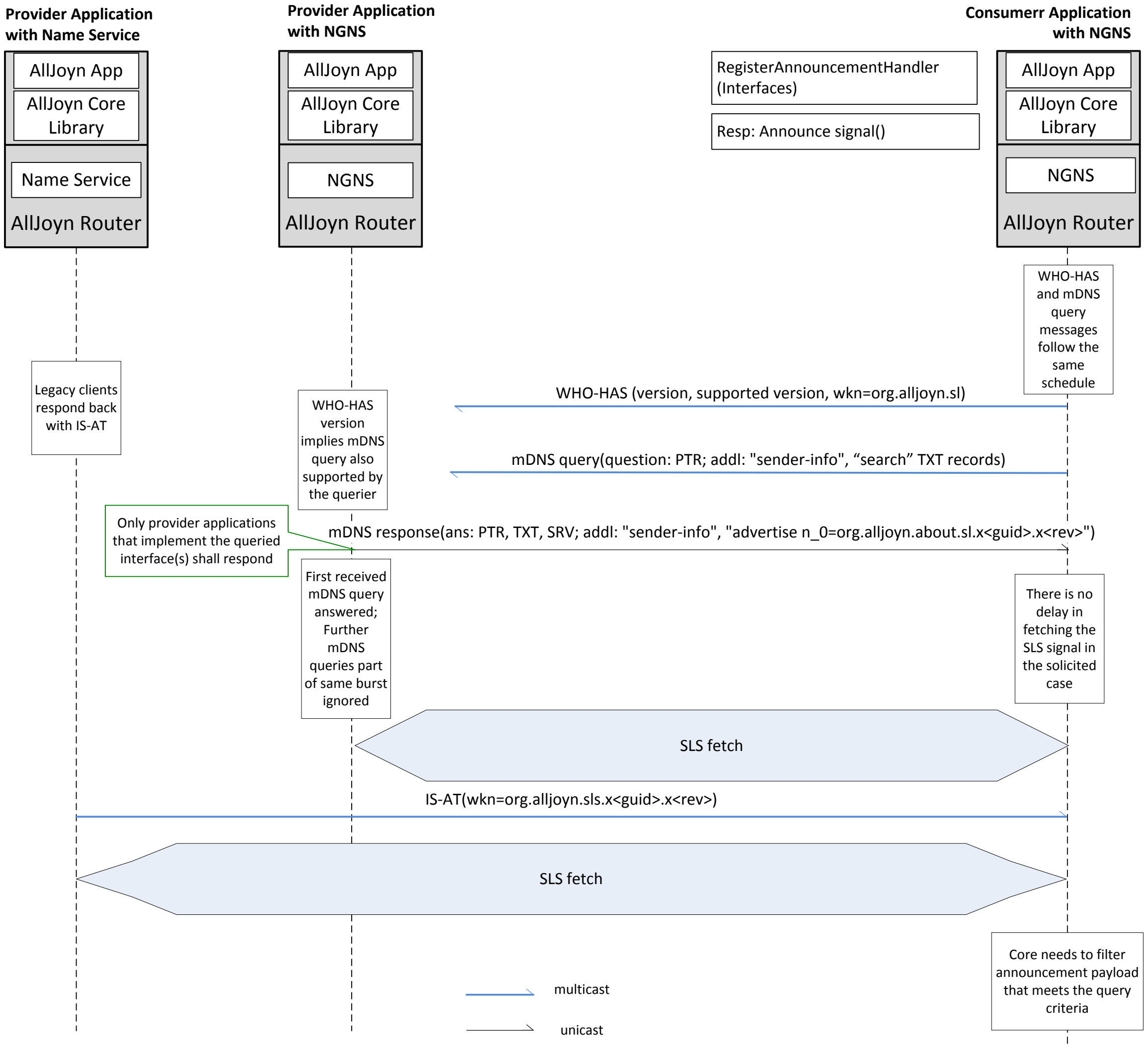

Provider Application with Name Service
Provider Application with NGNS
Consumerr Application with NGNS
AllJoyn App
AllJoyn Core Library
Name Service
NGNS
AllJoyn Router
RegisterAnnouncementHandler (Interfaces)
Resp: Announce signal()
WHO-HAS and mDNS query messages follow the same schedule
WHO-HAS (version, supported version, wkn=org.alljoyn.sl)
Legacy clients respond back with IS-AT
WHO-HAS version implies mDNS query also supported by the querier
mDNS query(question: PTR; addl: "sender-info", "search" TXT records)
Only provider applications that implement the queried interface(s) shall respond
mDNS response(ans: PTR, TXT, SRV; addl: "sender-info", "advertise n_0=org.alljoyn.about.sl.x<guid>.x<rev>")
First received mDNS query answered; Further mDNS queries part of same burst ignored
There is no delay in fetching the SLS signal in the solicited case
SLS fetch
IS-AT(wkn=org.alljoyn.sls.x<guid>.x<rev>)
SLS fetch
Core needs to filter announcement payload that meets the query criteria
multicast
unicast

Figure 4-19. Interface query (NGNS consumer app; NGNS and Name Service provider apps)

4.6.2.3 Pending AllJoyn interface names query; provider apps arrive later

This message sequence describes the scenario when there is a pending query (i.e., the transmission schedule has expired) but the Announce signal handler is still registered.

The main steps for the message sequence are described below:

1. The message sequence is initiated by the consumer application registering the announce handler (by calling RegisterAnnounceHandler) and providing a set of AllJoyn interfaces for discovery.
2. The NGNS sends DNS-SD based query messages over mDNS, and populates the search TXT record in the Additional section based on the AllJoyn interfaces being discovered.

3. The NGNS sends WHO-HAS discovery messages with WKN=org.alljoyn.sl.
4. The query schedule for mDNS messages and WHO-HAS message expires
5. Upon arrival of the provider application on the AllJoyn network, the NGNS sends unsolicited DNS-SD response messages with the sessionless signal well-known names and also advertises the sessionless signal well-known names via IS-AT messages.
6. Upon joining the AllJoyn network, the Name Service provider application advertises the sessionless signal well-known name via IS-AT messages.
7. The consumer AllJoyn router fetches the sessionless signals from the provider apps and performs filtering; the Announce signal is sent to the consumer application if there is a match.

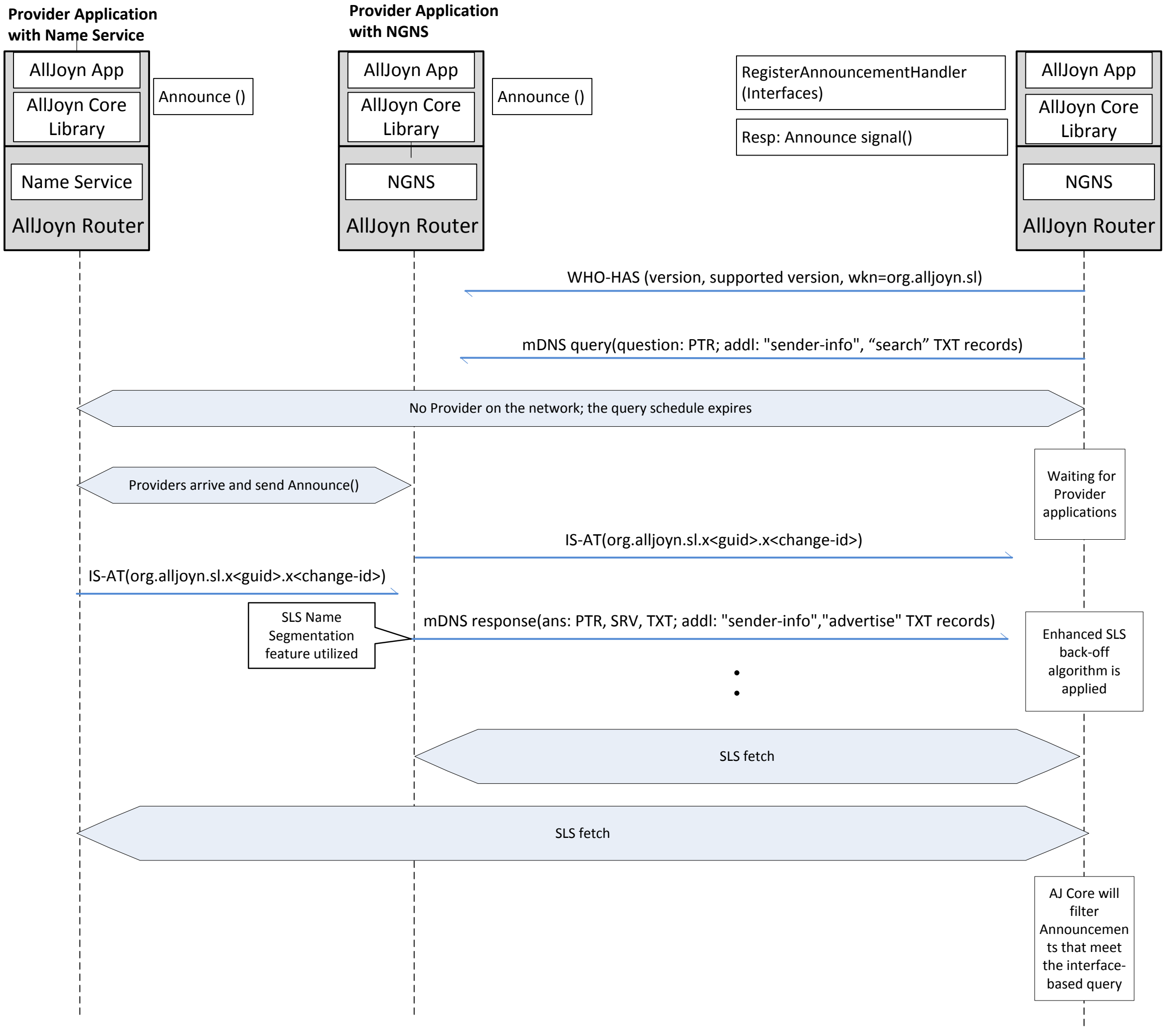

Provider Application with Name Service
AllJoyn App
AllJoyn Core Library
Name Service
AllJoyn Router
Announce ()
Provider Application with NGNS
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
Announce ()
RegisterAnnouncementHandler (Interfaces)
Resp: Announce signal()
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
WHO-HAS (version, supported version, wkn=org.alljoyn.sl)
mDNS query(question: PTR; addl: "sender-info", "search" TXT records)
No Provider on the network; the query schedule expires
Providers arrive and send Announce()
Waiting for Provider applications
IS-AT(org.alljoyn.sl.x<guid>.x<change-id>)
IS-AT(org.alljoyn.sl.x<guid>.x<change-id>)
SLS Name Segmentation feature utilized
mDNS response(ans: PTR, SRV, TXT; addl: "sender-info","advertise" TXT records)
Enhanced SLS back-off algorithm is applied
SLS fetch
SLS fetch
AJ Core will filter Announcements that meet the interface-based query

Figure 4-20. Pending AllJoyn interface query (NGNS consumer app, NGNS and Name Service provider apps)

4.6.3 Cancel advertisement

4.6.3.1 NGNS provider app, NGNS and Name Service consumer apps

The main steps for the message sequence are described below.

1. The provider application calls CancelAdvertiseName().
2. The NGNS sends both IS-AT and DNS-SD response message. Note that advertise TXT record in the mDNS message has TTL set to 0.
3. The consumer application receives LostAdvertisedName() upon receipt of the cancel advertisement discovery message.

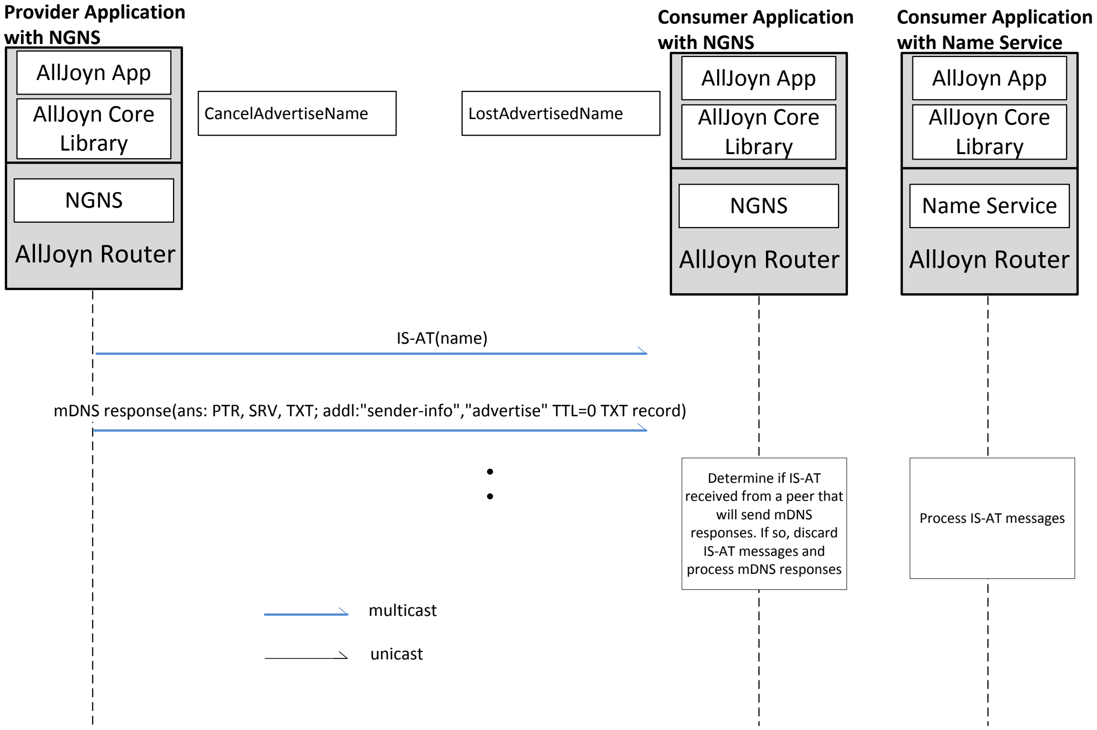

Provider Application with NGNS
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
CancelAdvertiseName
LostAdvertisedName
Consumer Application with NGNS
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
Consumer Application with Name Service
AllJoyn App
AllJoyn Core Library
Name Service
AllJoyn Router
IS-AT(name)
mDNS response(ans: PTR, SRV, TXT; addl:"sender-info","advertise" TTL=0 TXT record)
Determine if IS-AT received from a peer that will send mDNS responses. If so, discard IS-AT messages and process mDNS responses
Process IS-AT messages
multicast
unicast

Figure 4-21. Cancel advertised name (NGNS consumer app, NGNS and Name Service provider apps)

4.6.4 Presence

4.6.4.1 NGNS consumer app and NGNS provider app

Two modes of the Presence API is supported: synchronous and asynchronous mode. From the perspective of wire protocol, the message sequence is identical.

In the 14.02 release, presence is validated by the receipt of the IS-AT messages for a name; and three successive losses of IS-AT messages trigger a LostAdvertisedName()

to the consumer application. Since the delay was not tolerable for most applications, the presence was redesigned in the 14.06 release. A consumer application can initiate the presence of a name that was previously discovered using the newly introduced Ping API.

- If the discovered name is connected to a 14.06 AllJoyn router, the presence message sequence is initiated.
- If the discovered name is connected to a 14.02 AllJoyn router, the API invocation returns an error.

The main steps for the message sequence are described below.

1. The consumer application initiates a presence check for the name by invoking the Ping API.
2. The AllJoyn router returns the unimplemented error code if the name being pinged is connected to a 14.02 AllJoyn router at the time of discovery; else, the message sequence continues.
3. If there is an entry for the name in the AllJoyn routing table, then mDNS message is sent over unicast to check the presence state.
4. Upon receipt of the mDNS message, the AllJoyn router checks the presence state for the name and sends an mDNS response message over unicast. The presence check is performed using D-Bus Ping method call.

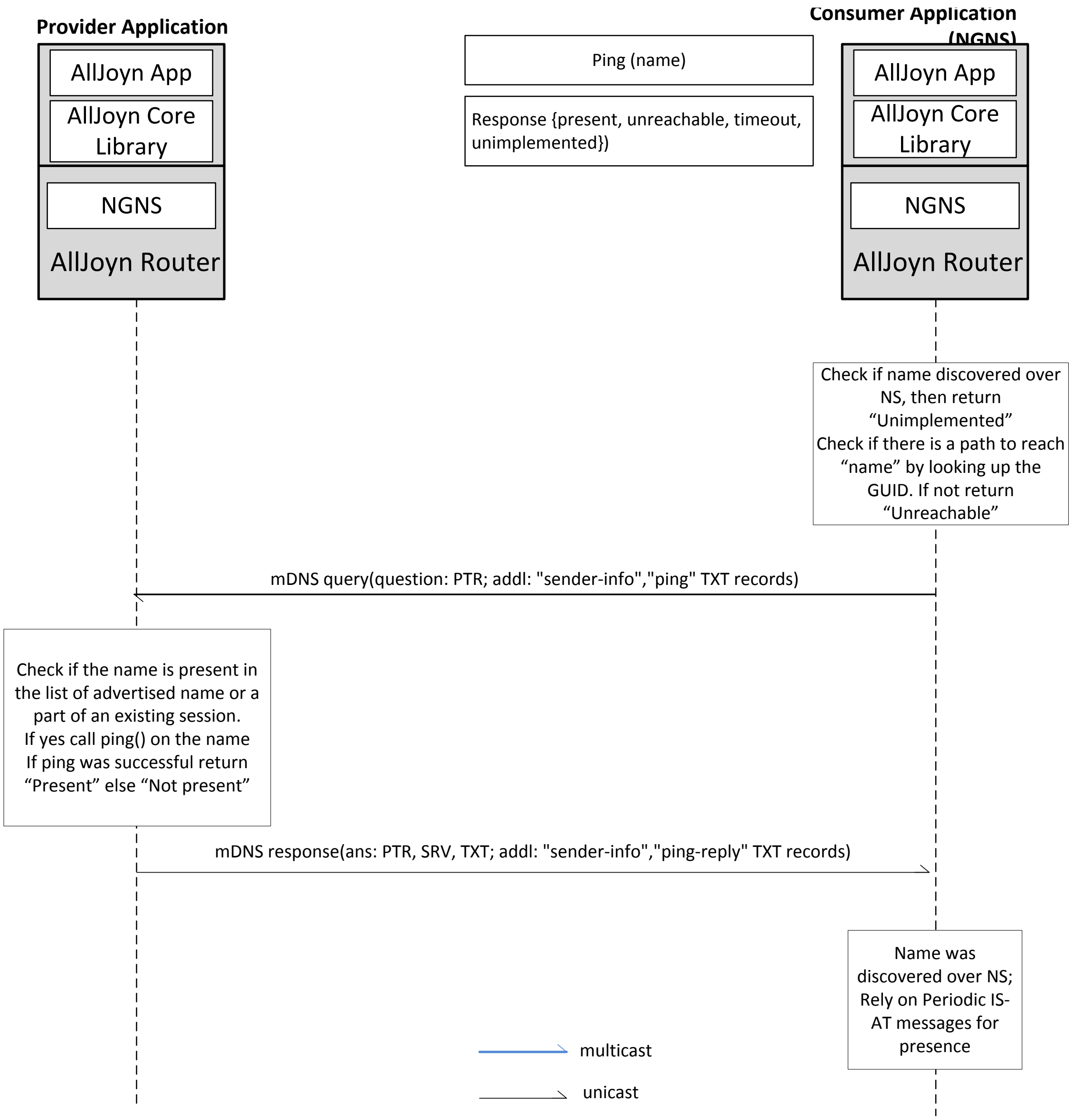

Provider Application
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
Ping (name)
Response {present, unreachable, timeout, unimplemented})
Consumer Application (NGNS)
AllJoyn App
AllJoyn Core Library
NGNS
AllJoyn Router
Check if name discovered over NS, then return "Unimplemented"
Check if there is a path to reach "name" by looking up the GUID. If not return "Unreachable"
mDNS query(question: PTR; addl: "sender-info","ping" TXT records)
Check if the name is present in the list of advertised name or a part of an existing session.
If yes call ping() on the name
If ping was successful return "Present" else "Not present"
mDNS response(ans: PTR, SRV, TXT; addl: "sender-info","ping-reply" TXT records)
Name was discovered over NS; Rely on Periodic IS-AT messages for presence
multicast
unicast

Figure 4-22. Ping API called by consumer application over NGNS (NGNS consumer app and NGNS provider app)

4.6.4.2 Legacy presence with NGNS consumer app and Name Service provider app

If a name that was discovered is connected to a 14.02 AllJoyn router, the new Ping API message sequence is not supported. If the Ping API returns not implemented error code, the consumer application must issue FindAdvertisedName() with the discovered name so that presence for that name can be initiated.

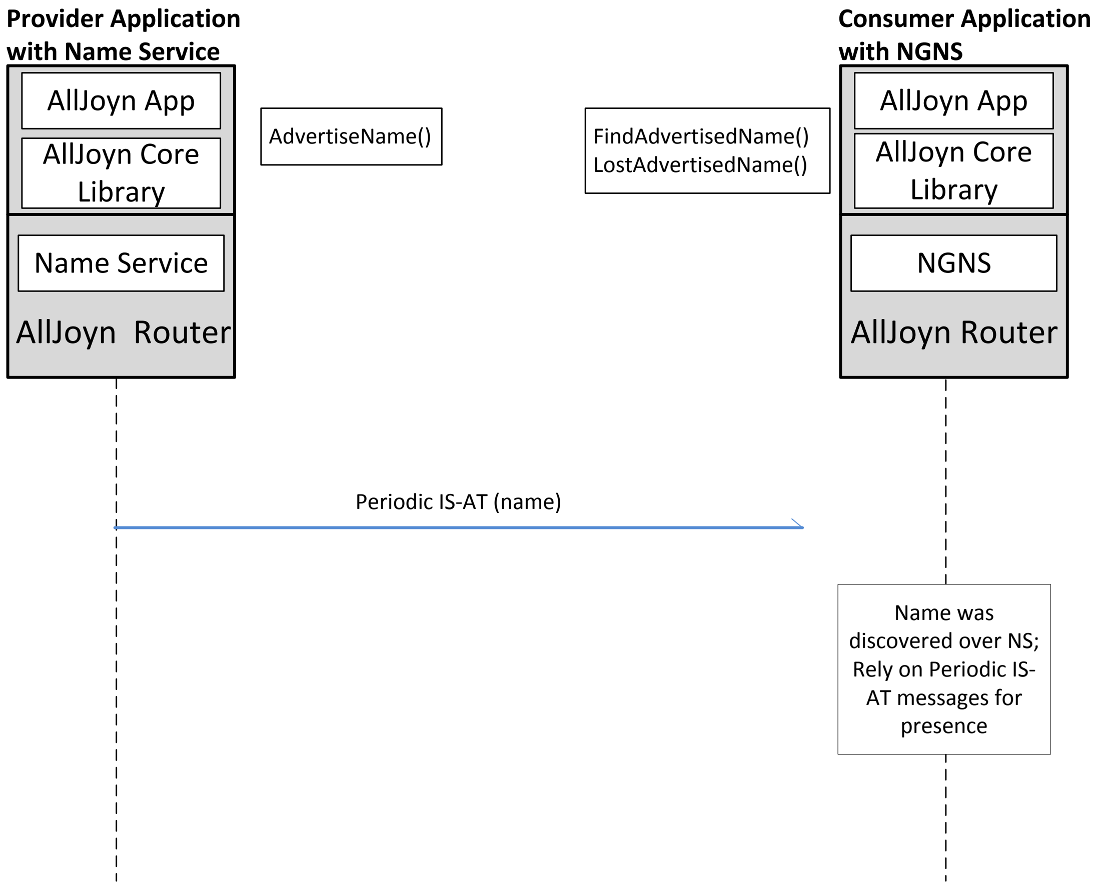


Figure 4-23. Reverting to legacy presence (NGNS consumer app and Name Service provider app)

4.7 DNS-SD message format

See section 4.5.3.2 for information on how the AllJoyn framework makes use of the DNS-SD protocol.

The AllJoyn discovery process is based on the DNS-SD and the message format is captured below.

NOTE

<guid> in the resource records refers to the AllJoyn router’s GUID. In addition, the presence of specific records in a given message is specified in the NGNS message sequences capture above while the tables below show all the possible records types that can be present in the query or response messages:

4.7.1 DNS-SD query

4.7.1.1 Question

Table 4-8: DNS-SD query: question format

Name	Type	Record-specific data
■ alljoyn._udp.local. ■ alljoyn._tcp.local.	PTR	The service name is alljoyn as allocated through IANA. In the 14.06 release, the protocol used in the service description is TCP. When UDP transport is supported in future, the protocol for service name will be UDP. The discovery scope is the local network.

4.7.1.2 Additional section

Table 4-9: DNS-SD query: Additional section

Name	Type	Record-specific data
search.<guid>.local.	TXT	Captures the well-known names or interfaces that are being searched. The key notation is as follows: ■ txtvrs=0; this represents version of the TXT record ■ n_1, n_2, etc., if multiple well-known names are present, they are logically ANDed; n_# is the key for well-known names. ■ i_1, i_2, etc., if multiple interface names are being queried. If multiple interface names are present, they are logically ANDed; i_# is the key for interface names. ■ Since the APIs for name-based and interface-based query are different, the search record has either name keys or interface keys. If the consumer application intends to perform logical OR operation for interface names, it must call the discovery API with interface name multiple times. Example: i_1 = org.alljoyn.About
sender-info.<guid>.local.	TXT	Captures additional data regarding the sender of the message. The following keys are sent: ■ txtvrs=0; represents the version of the TXT record ■ pv (protocol version): represents the discovery protocol version ■ IPv4 and UDPv4 address: represents the IPv4 address and UDP port ■ bid (burst identifier): represents the burst identifier
ping.<guid>.local.	TXT	Captures the names that are being pinged by the consumer application. The key notation is as follows: ■ txtvrs=0; represents version of the TXT record ■ n= the well-known name or the unique name Only one key can be present in the ping record.

4.7.2 DNS-SD response

4.7.2.1 Answer

Table 4-10: DNS-SD response message: Answer section

Name	Type	Record Specific Data
_alljoyn._tcp.local.	PTR	<guid>._alljoyn._tcp.local.
<guid>._alljoyn._tcp.local.	TXT	txtvrs=0 Except for text record version, there is no additional record
<guid>._alljoyn._tcp.local.	SRV	port, <guid>.local port reptesents TCP port number used for the router-router connection

4.7.2.2 Additional section

Table 4-11: DNS-SD response message: Additional section

Name	Type	Record Specific Data
advertise.<guid>.local.	TXT	Captures the well-known names that the provider application is advertising.The key notation is as follows: n_1, n_2, etc., if multiple well-known names are being advertised; n_# is the key for well-known names. For interface query response, the sessionless signal well-known name that is advertised is as follows: n_1=org.alljoyn.About.sl.y<guid>.x<latest change_id>
sender-info.<guid>.local.	TXT	Captures additional data regarding the sender of the message. The following keys are sent: ■ txtvrs=0; represents version of the TXT record ■ pv (protocol version): represents the discovery protocol version ■ IPv4 and UDPv4 address: represents the IPv4 address and UDP port ■ bid (burst identifier): represents the burst identifier
Ping-reply.<guid>.local.	TXT	Captures the names that are being pinged by the consumer application. The key notation is as follows: ■ txtvrs=0; represents version of the TXT record ■ n= well-known name or unique name ■ replycode = reply code as returned by the router
<guid>.local	A	This resource record sends IPv4 address. It is present in response messages for discovery.

4.7.3 NGNS configuration parameters

Table 4-12 lists the NGNS configuration parameters.

Table 4-12: NGNS configuration paramters

Parameter	Default value	Range	Description
EnableLegacyNS	true	Boolean	Specifies the backward compatibility behavior with respect to legacy Name Service.

4.8 Discovery usage guidelines

Although the AllJoyn system supports both the name-based and announcement-based discovery, the preferred and recommended method for discovering services in an AllJoyn IoE network is the announcement-based discovery.

The name-based service discovery process can be used for app-to-app based discovery, where both provider and consumer applications are aware of the well-known name. This discovery process is also used for sessionless signals and to discover the AllJoyn router for the thin app. Table 4-13 summarizes usage guidelines for the two AJ discovery methods.

Table 4-13. AllJoyn discovery method usage guidelines

Name-based discovery usage	Announcement-based discovery usage
■ App-to-app discovery ■ Sessionless signals ■ AllJoyn router discovery for thin apps	AllJoyn service interfaces discovery on the AllJoyn network

5 AllJoyn Session

5.1 Overview

After an AllJoyn consumer has discovered some desired services offered by provider devices, the next step is to establish an AllJoyn session with the provider to consume those services. An AllJoyn session is a logical connection between consumer and provider applications that allows these applications to communicate with each other and exchange data. A provider application creates an AllJoyn session and waits for consumer applications to join the session. The application that creates the session is the owner of the session (session host), and other applications are termed as joiners in that session.

On the provider side, the app binds a session port with the AllJoyn core library, specifying a list of session options (e.g., transports, session type, etc.) and then listens for consumers to join the session. The session port is typically known ahead of time to both the consumer and the provider app. Alternately, the session port can be discovered via the Announcement message received from that provider.

On the consumer side, the app requests the AllJoyn bus to join the session with a given provider app by specifying the session port, well-known name/unique name for the service, and session options (transport, session type, etc.). After this, the AllJoyn router initiates a session establishment flow between the consumer and the provider apps. A unique session ID gets assigned by the provider to the AllJoyn session after the first client joins the session. The session ID is sent back to the consumer app and should be used for subsequent communication with the provider.

Figure 5-1 shows the high-level architecture for the AllJoyn session establishment.

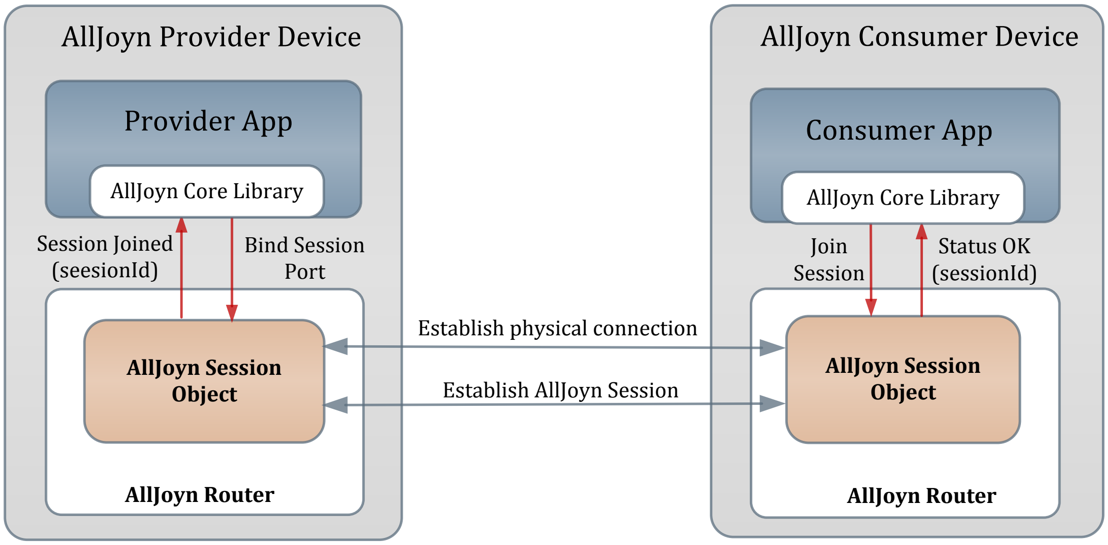


Figure 5-1. AllJoyn session establishment architecture

The provider app binds a session port with the AllJoyn router for a given service. The combination of (well-known name/unique name and session port) uniquely identifies the endpoint. The consumer app starts a join session with the (well-known name/unique name + session port number) of the provider app. The AllJoyn router on the consumer side establishes a physical connection with the provider AllJoyn router based on already discovered information as part of the discovery step. Currently, this involves establishing a TCP connection over Wi-Fi. The physical connection between the consumer and provider AllJoyn routers can be a single-hop or multi-hop connection based on the underlying transport.

After physical connection is established, the consumer AllJoyn router starts the session establishment with the provider side. The provider AllJoyn router assigns a unique session ID for the session and also creates a session map storing the relevant session information. Once the session is established, a Session Joined callback is sent to the provider app with generated session ID. The consumer app receives a Status OK response for the join session call that includes the session ID. A session map also gets created on the consumer AllJoyn router side, storing session details.

5.2 Types of sessions

An AllJoyn session can be categorized into different types based on the allowed number of participants in the session or the data encapsulation option used over the session.

The AllJoyn system supports the following types of session based on allowed number of participants:

- Point-to-point session: An AllJoyn session with a single consumer (joiner) and single provider (session host) endpoints participates in the session. When either participant leaves the session, the point-to-point session ends. A SessionLost indication is sent to the remaining participant.
- Multi-Point Session: An AllJoyn session allows more than two participants. Such a session involves a provider app (session host app) and one or more consumer apps (joiner apps) participating in the same session. A multi-point session can be joined multiple times to form a single session with multiple (greater than 2) endpoints. New consumers can join a multi-point session after the session has been created, and existing consumers can leave a multi-point session. All participants in a multi-point session can communicate to each other.

In a multi-point session, the AllJoyn framework does not mandate that all communications go through session host; participants can directly talk to each other if allowed by the underlying transport. Certain transports such as Bluetooth may require all the communication to go through the session host based on the topology. Similar to a point-to-point session, a multi-point session ends when two participants are left and one of them leaves the session. A SessionLost indication is sent to the remaining participant.

Figure 5-2 depicts point-to-point and multi-point AllJoyn sessions with the multi-point session showing four participants.

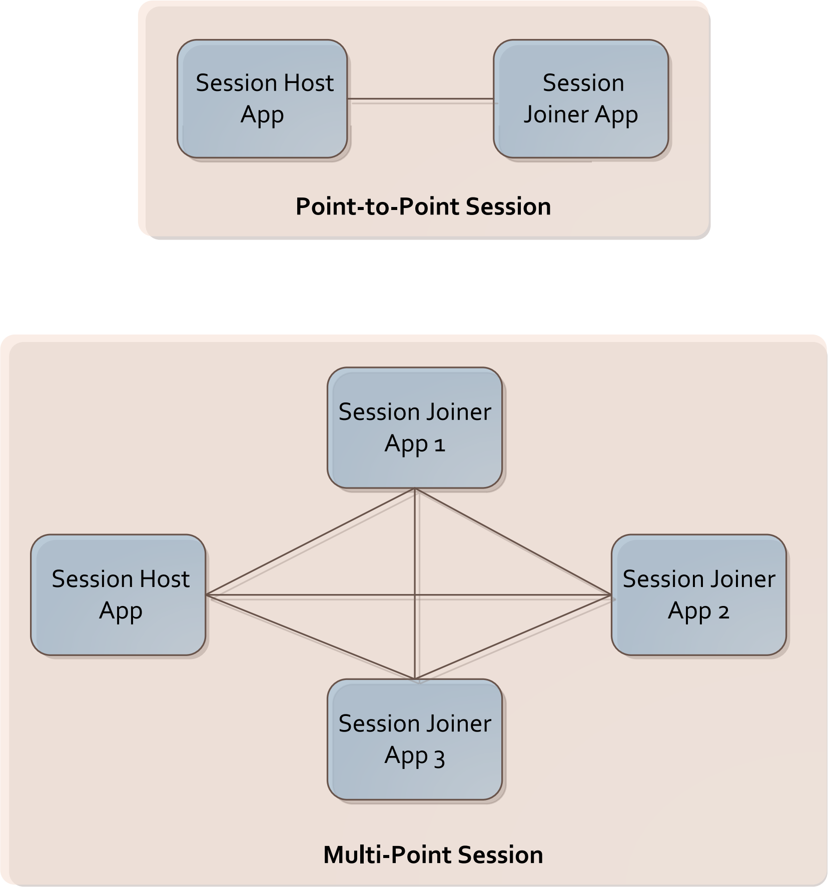


Figure 5-2. AllJoyn point-to-point and multi-point session examples

5.2.1 Raw session

In the AllJoyn system, typical data exchange between peer nodes occurs in the form of enhanced D-Bus messages. However in some scenarios, the overhead associated with D-Bus messages may not be desirable. In such cases, raw data can be exchanged between nodes using what is called an AllJoyn raw session.

An AllJoyn raw session is used to exchange raw data between endpoints using an underlying physical connection (e.g., TCP/UDP socket-based communication). A raw session does not carry D-Bus encapsulated messages like a regular AllJoyn session. Instead, a raw session carries unencapsulated raw data directly sent over TCP/UDP sockets. A raw session can only be a point-to-point session.

NOTE

The raw session feature is only supported on the AllJoyn standard client and is not supported on thin app. This feature is being deprecated and it is recommended that developers not to use the raw session feature.

5.3 Session establishment

Figure 5-3 captures the AllJoyn session establishment message flow for a point-to-point session.

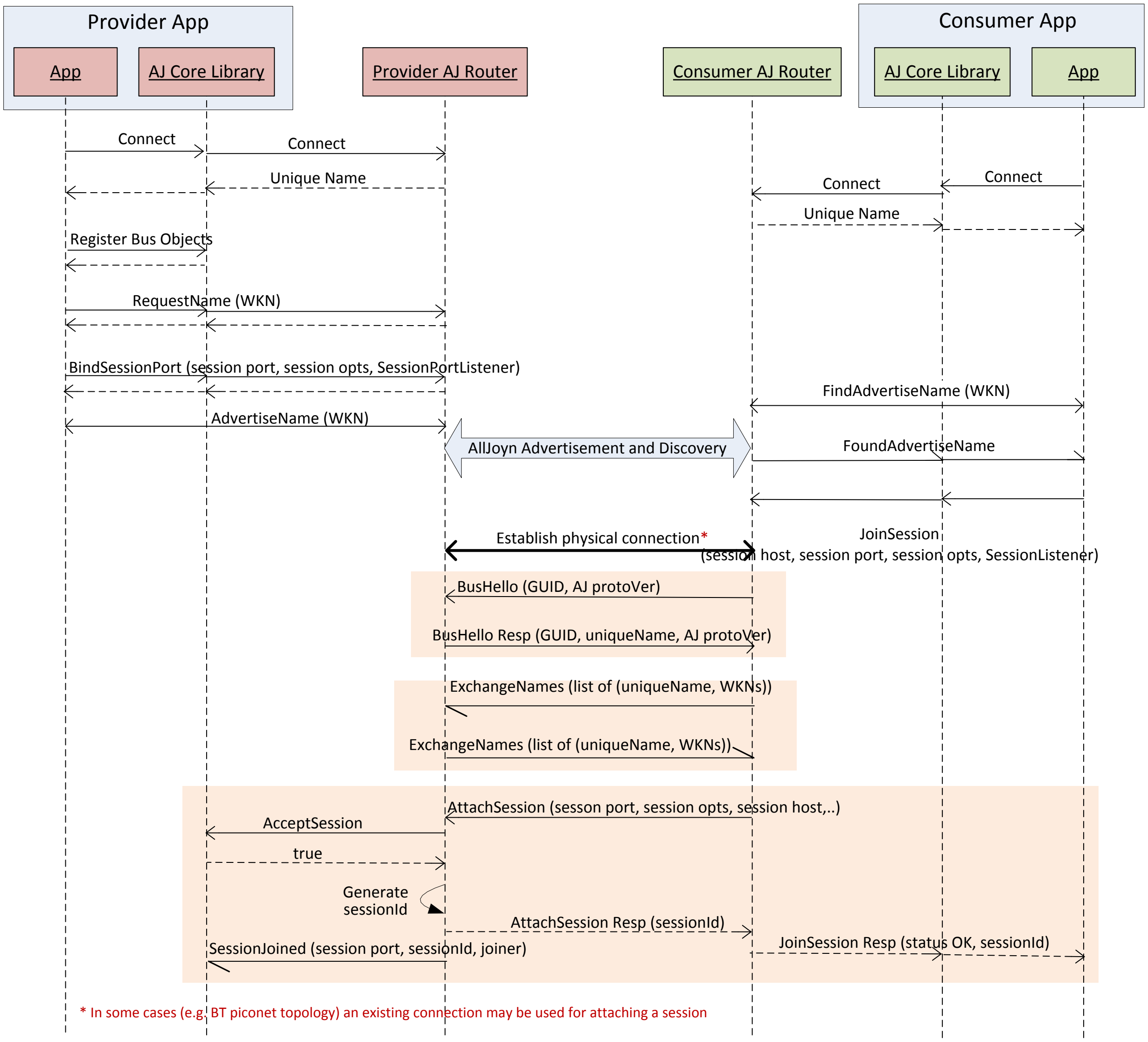

Provider App
App
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer App
AJ Core Library
App
Connect
Connect
Unique Name
Connect
Connect
Unique Name
Register Bus Objects
RequestName (WKN)
BindSessionPort (session port, session opts, SessionPortListener)
FindAdvertiseName (WKN)
AdvertiseName (WKN)
AllJoyn Advertisement and Discovery
FoundAdvertiseName
Establish physical connection*
JoinSession
(session host, session port, session opts, SessionListener)
BusHello (GUID, AJ protoVer)
BusHello Resp (GUID, uniqueName, AJ protoVer)
ExchangeNames (list of (uniqueName, WKNs))
ExchangeNames (list of (uniqueName, WKNs))
AttachSession (sesson port, session opts, session host,..)
AcceptSession
true
Generate
sessionId
AttachSession Resp (sessionId)
JoinSession Resp (status OK, sessionId)
SessionJoined (session port, sessionId, joiner)
* In some cases (e.g. BT piconet topology) an existing connection may be used for attaching a session

Figure 5-3. AllJoyn session - establishing a point-to-point session

The message flow steps are described below.

1. Both the provider and consumer apps connect with their respective AllJoyn routers via the AllJoyn core library and get a unique name assigned.
2. The provider app registers service Bus Objects with the AllJoyn core library.
3. The provider app requests a well-known name with the AllJoyn router via the AllJoyn core library.

4. The provider app binds a session port with the AllJoyn router via the AllJoyn core library's BindSessionPort API. This call specifies a session port, session options, and a SessionPortListener for the session.
5. The provider app advertises the well-known name with the AllJoyn router via the AllJoyn core library's AdvertiseName API.
6. The provider side AllJoyn router initiates a transport-specific advertisement for the well-known name.
7. The consumer app initiates a discovery for the same well-known name via the AllJoyn core library's FIndAdvertiseName API.
8. The consumer side AllJoyn router performs transport-specific AllJoyn discovery to find the desired well-known name. It sends a FoundAdvertiseName signal to the consumer app.
9. The consumer app initiates joining the session with the provider via the JoinSession API. This call specifies unique name of session host, session port, desired session options, and a SessionListner.
10. The consumer side AllJoyn router establishes a physical channel with the provider side AllJoyn router (it sets up a TCP connections between the two AllJoyn buses).
11. Once a connection is set up between the two AllJoyn buses, the consumer AllJoyn router initiates a BusHello message to send its bus GUID and AllJoyn protocol version. The provider AllJoyn router responds with its GUID, AllJoyn protocol version, and unique name.
12. The consumer and provider AllJoyn routers send out ExchangeNames signals to exchange the set of known unique names and well-known names.
13. The consumer AllJoyn router invokes the AttachSession method call at the provider AllJoyn router to join the session. This call specifies session port, session options, and unique name/well-known name of the session host among other parameters.
14. The provider AllJoyn router invokes an AcceptSession method call with the provider app which returns 'true' if the session gets accepted.
15. The provider AllJoyn router generates a unique sessionId for this session. It sends back an AttachSession response message to the consumer AllJoyn router providing the sessionId.
16. The provider AllJoyn router sends a SessionJoined signal to the provider app specifying the sessionId.
17. After receiving the AttachSession response, the consumer AllJoyn router sends a JoinSession response message to the app with an OK status and provides the session Id.

5.3.1 Use cases

The following use cases illustrate various AllJoyn session scenarios:

- Establish a multi-point session
- Consumer joins an existing multi-point session

- Consumer leaves a point-to-point Session
- Consumer leaves a multi-point session with more than 2 participants
- Provider unbinds a session port
- Incompatible session options

5.3.1.1 Establish a multi-point session

Figure 5-4 captures the session establishment message flow for a multi-point session between two participants.

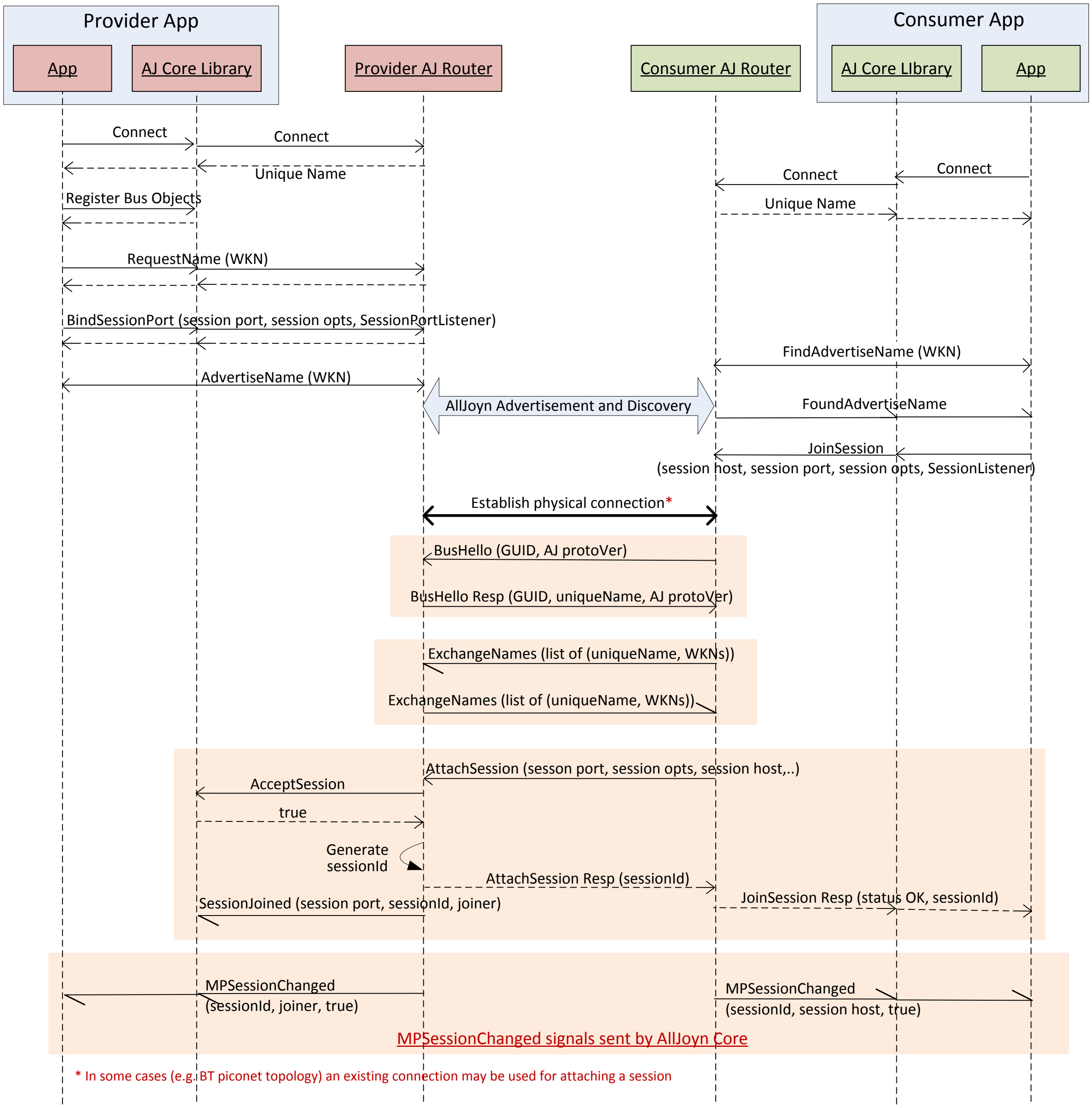

Provider App
App
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer App
AJ Core LIbrary
App
Connect
Connect
Unique Name
Connect
Connect
Register Bus Objects
Unique Name
RequestName (WKN)
BindSessionPort (session port, session opts, SessionPortListener)
FindAdvertiseName (WKN)
AdvertiseName (WKN)
AllJoyn Advertisement and Discovery
FoundAdvertiseName
JoinSession
(session host, session port, session opts, SessionListener)
Establish physical connection*
BusHello (GUID, AJ protoVer)
BusHello Resp (GUID, uniqueName, AJ protoVer)
ExchangeNames (list of (uniqueName, WKNs))
ExchangeNames (list of (uniqueName, WKNs))
AttachSession (sesson port, session opts, session host,..)
AcceptSession
true
Generate sessionId
AttachSession Resp (sessionId)
JoinSession Resp (status OK, sessionId)
SessionJoined (session port, sessionId, joiner)
MPSessionChanged
(sessionId, joiner, true)
MPSessionChanged
(sessionId, session host, true)
MPSessionChanged signals sent by AllJoyn Core
* In some cases (e.g. BT piconet topology) an existing connection may be used for attaching a session

Figure 5-4. AllJoyn Session - Establishing Multi-point Session

A multi-point session follows same message flow as the point-to-point session with the additional step of sending out the MPSessionChanged signal from the AllJoyn router to the application indicating new participant. This signal specifies the sessionId, the unique name/well-known name of the participant, and a flag to indicate whether the participant was added.

5.3.1.2 Consumer joins an existing multi-point session

Figure 5-5 captures the message flow for the scenario where a new consumer joins an existing multi-point session.

In a multi-point session, the new joiner is responsible for notifying existing participants (other than session host) of the newly added member to the session. This is so that existing members can update their session routing information to include the new joiner, and future session messages can be routed appropriately. To achieve this, the new member invokes an AttachSession with all existing members. This results in existing members adding the new joiner to their session-related tables.

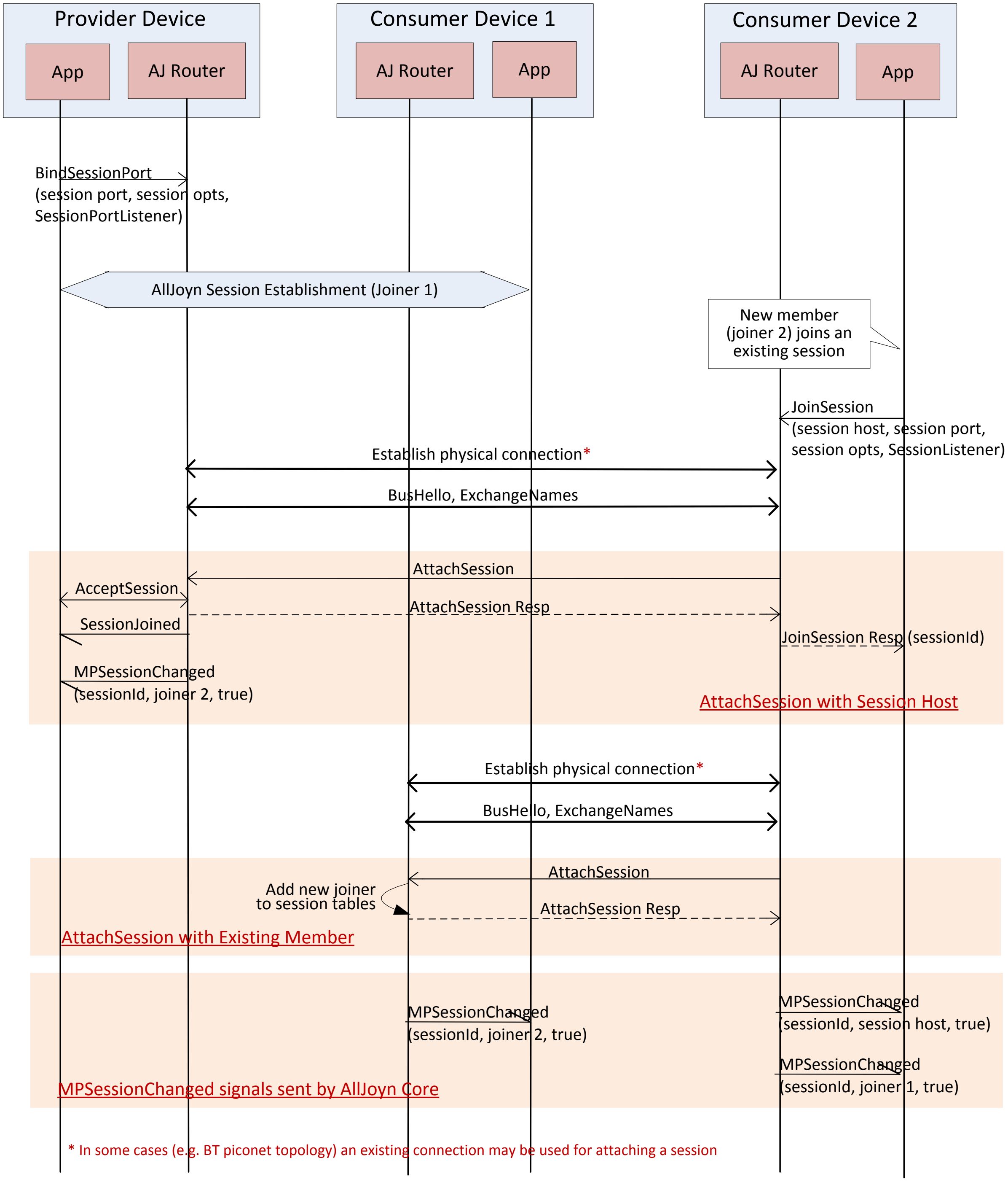

Provider Device
App
AJ Router
Consumer Device 1
AJ Router
App
Consumer Device 2
AJ Router
App
BindSessionPort
(session port, session opts,
SessionPortListener)
AllJoyn Session Establishment (Joiner 1)
New member
(joiner 2) joins an
existing session
JoinSession
(session host, session port,
session opts, SessionListener)
Establish physical connection*
BusHello, ExchangeNames
AttachSession
AcceptSession
AttachSession Resp
SessionJoined
JoinSession Resp (sessionId)
MPSessionChanged
(sessionId, joiner 2, true)
AttachSession with Session Host
Establish physical connection*
BusHello, ExchangeNames
AttachSession
Add new joiner
to session tables
AttachSession Resp
AttachSession with Existing Member
MPSessionChanged
(sessionId, joiner 2, true)
MPSessionChanged
(sessionId, session host, true)
MPSessionChanged
(sessionId, joiner 1, true)
MPSessionChanged signals sent by AllJoyn Core
* In some cases (e.g. BT piconet topology) an existing connection may be used for attaching a session

Figure 5-5. AllJoyn session - consumer joins a multi-point session

The message flow steps are described below.

1. The provider app binds a session port with the AllJoyn router via the AllJoyn core library's BindSessionPort API to expose its services.

2. The AllJoyn session establishment steps occur between joiner 1 (consumer 1) and the session host (provider) to establish a multi-point session as captured in section 5.3.1.1.
3. Consumer 2 (joiner 2) wants to join the existing multi-point session, and initiates a JoinSession call with its AllJoyn router.
4. The AllJoyn session establishment steps occur between joiner 2 and the session host to add this joiner to the existing multi-point session.
5. An MPSessionChanged signal is sent out to the session host app informing it of new joiner in the session.
6. Joiner 2 receives the set of existing members for the multi-point session from the session host as part of the AttachSession response.
7. Joiner 2 initiates an AttachSession with every received member of the session (except the session host, which it just did using the AttachSession).
8. Joiner 1 receives AttachSession from joiner 2 and updates its session-related tables to add joiner 2.
9. The AllJoyn router on joiner 1 sends out an MPSessionChanged signal to the app, indicating a newly added member to the multi-point session.
10. Joiner 2 also sends out separate MPSessionChanged signal to the app for each existing member of the session.

5.3.1.3 Consumer leaves a point-to-point session

Figure 5-6 captures the message flow for the scenario where a consumer leaves an existing point-to-point session. The same message flow is also applicable for the scenario when a consumer leaves a multi-point session with only two participants.

When a participant leaves a point-to-point session or a multi-point session with only two participants, the session ends and is removed from session tables of both the participants. A participant can leave a session by initiating a LeaveSession call with the AllJoyn router. This results in a DetachSession signal being delivered to the other member of the session. Receipt of this signal triggers clearing of sessionId and other session-related information from the session tables of that member. Whenever a session ends, a SessionLost signal is sent to the application.

NOTE

Either the joiner or the host of the session can leave a session. A similar message flow is applicable when a session host leaves the session.

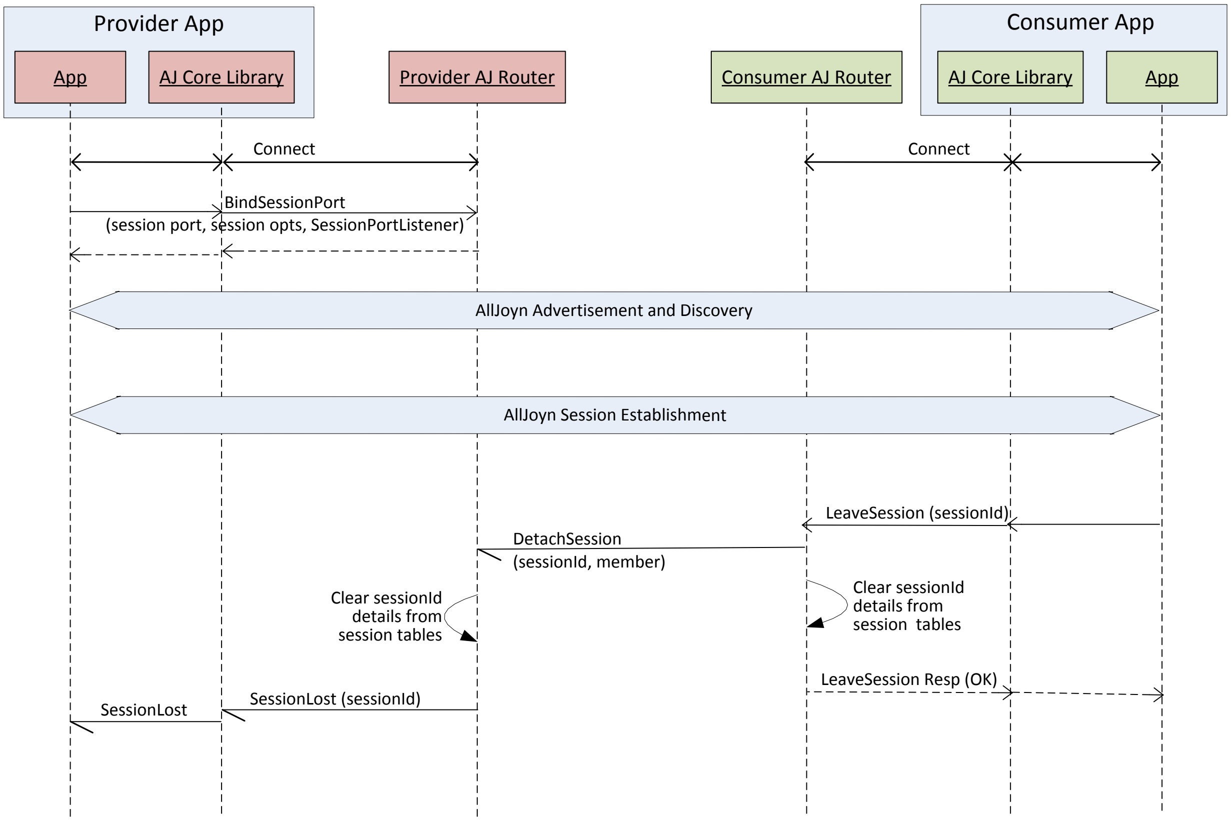


Figure 5-6. AllJoyn session - consumer leaves a point-to-point session

The message flow steps are described below.

1. The consumer app establishes a session with the session host.
2. The consumer app decides to leave the session. It invokes a LeaveSession API with the AllJoyn router via the AllJoyn core library. This call takes in the sessionId as input parameter.
3. The AllJoyn router generates a DetachSession signal specifying the sessionId and the member that is leaving the session. This signal is sent to the other member in the session.
4. After receiving the DetachSession signal, the AllJoyn router on the session host determines that it is the only member left in the session. As a result, it concludes that the session has ended and clears sessionId details from its session tables.
5. The AllJoyn router on the consumer side clears sessionId details from its session tables and sends a successful LeaveSession response to the application.
6. The AllJoyn router on the session host sends a SessionLost signal to the application indicating that the session has ended.

5.3.1.4 Consumer leaves a multi-point session

Figure 5-7 captures the message flow for the scenario where a consumer leaves a multi-point session with more than two participants.

In this scenario, the session continues with remaining participants even after a member leaves the session. The remaining participants update their session tables to remove the member that left the session.

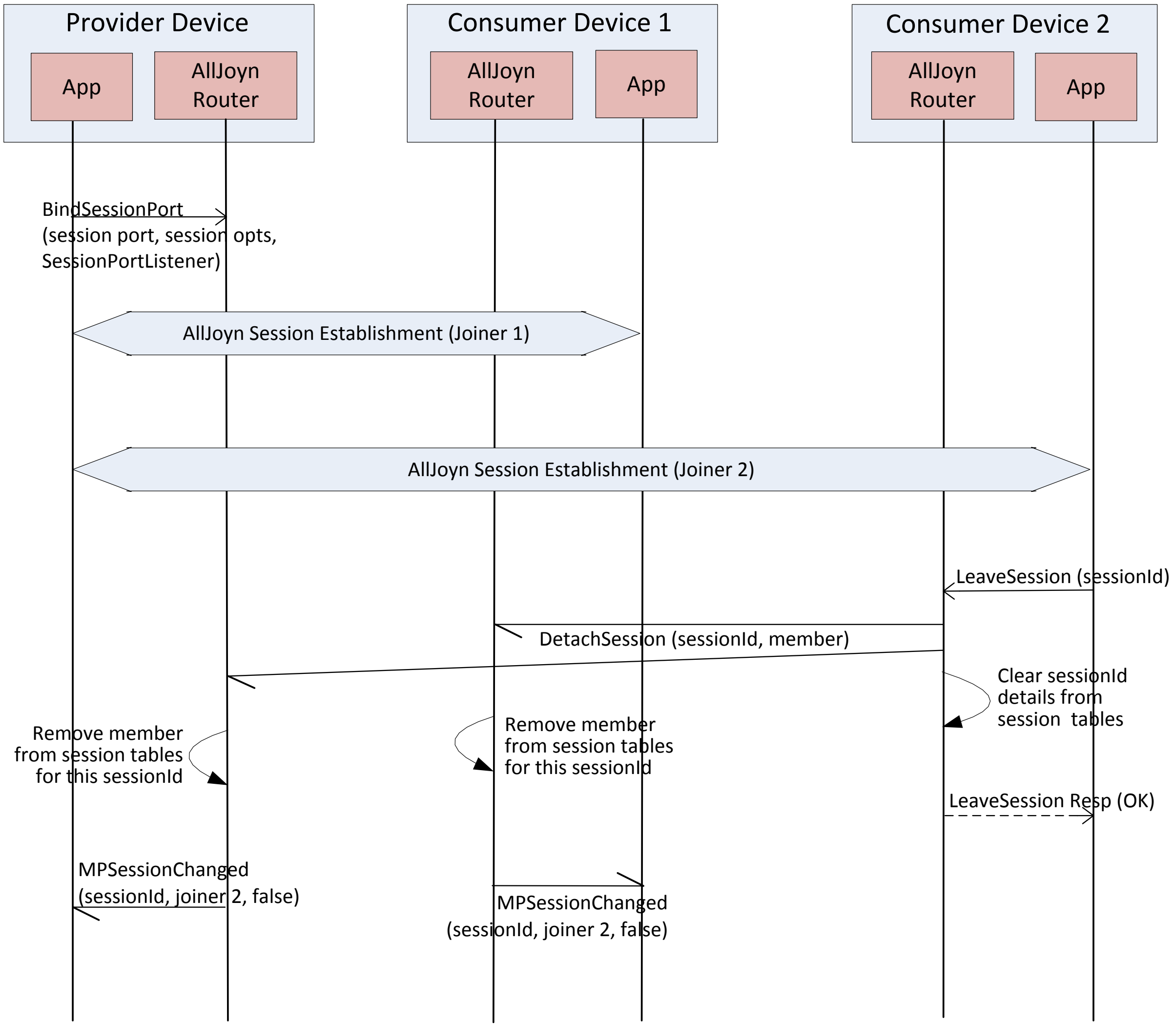

Provider Device
App
AllJoyn Router
Consumer Device 1
AllJoyn Router
App
Consumer Device 2
AllJoyn Router
App
BindSessionPort (session port, session opts, SessionPortListener)
AllJoyn Session Establishment (Joiner 1)
AllJoyn Session Establishment (Joiner 2)
LeaveSession (sessionId)
DetachSession (sessionId, member)
Clear sessionId details from session tables
Remove member from session tables for this sessionId
Remove member from session tables for this sessionId
LeaveSession Resp (OK)
MPSessionChanged (sessionId, joiner 2, false)
MPSessionChanged (sessionId, joiner 2, false)

Figure 5-7. AllJoyn session - consumer leaves a multi-point session

The message flow steps are described below.

1. Two consumer apps (joiner 1 and joiner 2) have joined in a single multi-point session with the provider.
2. Joiner 2 decides to leave the session. It invokes a LeaveSession API with the AllJoyn router, specifying the sessionId.

3. The AllJoyn router on joiner 2 generates a DetachSession signal, specifying the sessionId and the member that is leaving the session. This signal is sent as a session broadcast signal to all the other members in the session.
4. Upon receiving the DetachSession signal, the AllJoyn router involved in the multi-point session determines that there are two or more remaining participants in the session, meaning the session will continue to exist. As a result, it updates its session tables to remove the member received in the DetachSession signal for that sessionId. The AllJoyn router then sends an MPSessionChanged signal to the app indicating member deletion for that session. This logic is executed by the AllJoyn router for every remaining participant in the session.
5. The AllJoyn router on the member leaving the session clears sessionId details from its session tables and sends a successful LeaveSession response to the application.

5.3.1.5 Provider leaves a multi-point session

Figure 5-8 captures the message flow for the scenario where a provider (session host) leaves a multi-point session with more than two participants. In this case, the session continues to exist and the remaining participants can continue to communicate; however, no new participants can join the multi-point session.

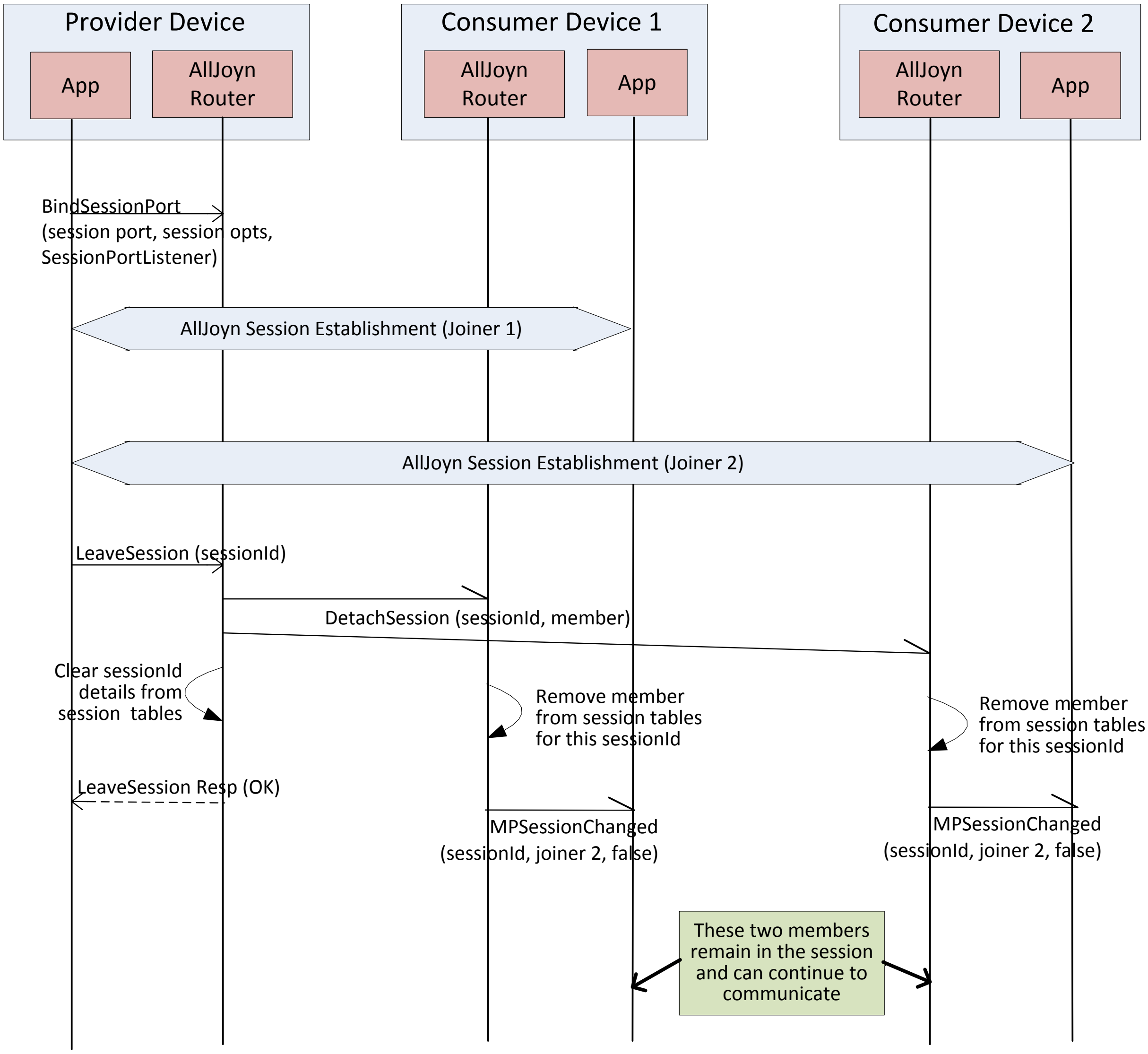

Provider Device
App
AllJoyn Router
Consumer Device 1
AllJoyn Router
App
Consumer Device 2
AllJoyn Router
App
BindSessionPort (session port, session opts, SessionPortListener)
AllJoyn Session Establishment (Joiner 1)
AllJoyn Session Establishment (Joiner 2)
LeaveSession (sessionId)
DetachSession (sessionId, member)
Clear sessionId details from session tables
Remove member from session tables for this sessionId
Remove member from session tables for this sessionId
LeaveSession Resp (OK)
MPSessionChanged (sessionId, joiner 2, false)
MPSessionChanged (sessionId, joiner 2, false)
These two members remain in the session and can continue to communicate

Figure 5-8. AllJoyn session – provider leaves a multi-point session

5.3.1.6 Provider unbinds a session port

The provider app can unbind a previously bound session port at any time. As a result, no new sessions can be established on that session port. Any existing sessions on that session port will continue and are not impacted. If there was any multi-point session on that session port, no new members can be added to that multi-point session.

5.3.1.7 Incompatible session options

Figure 5-9 captures the message flow for the scenario when the session options requested by the consumer is incompatible with session options specified by the provider.

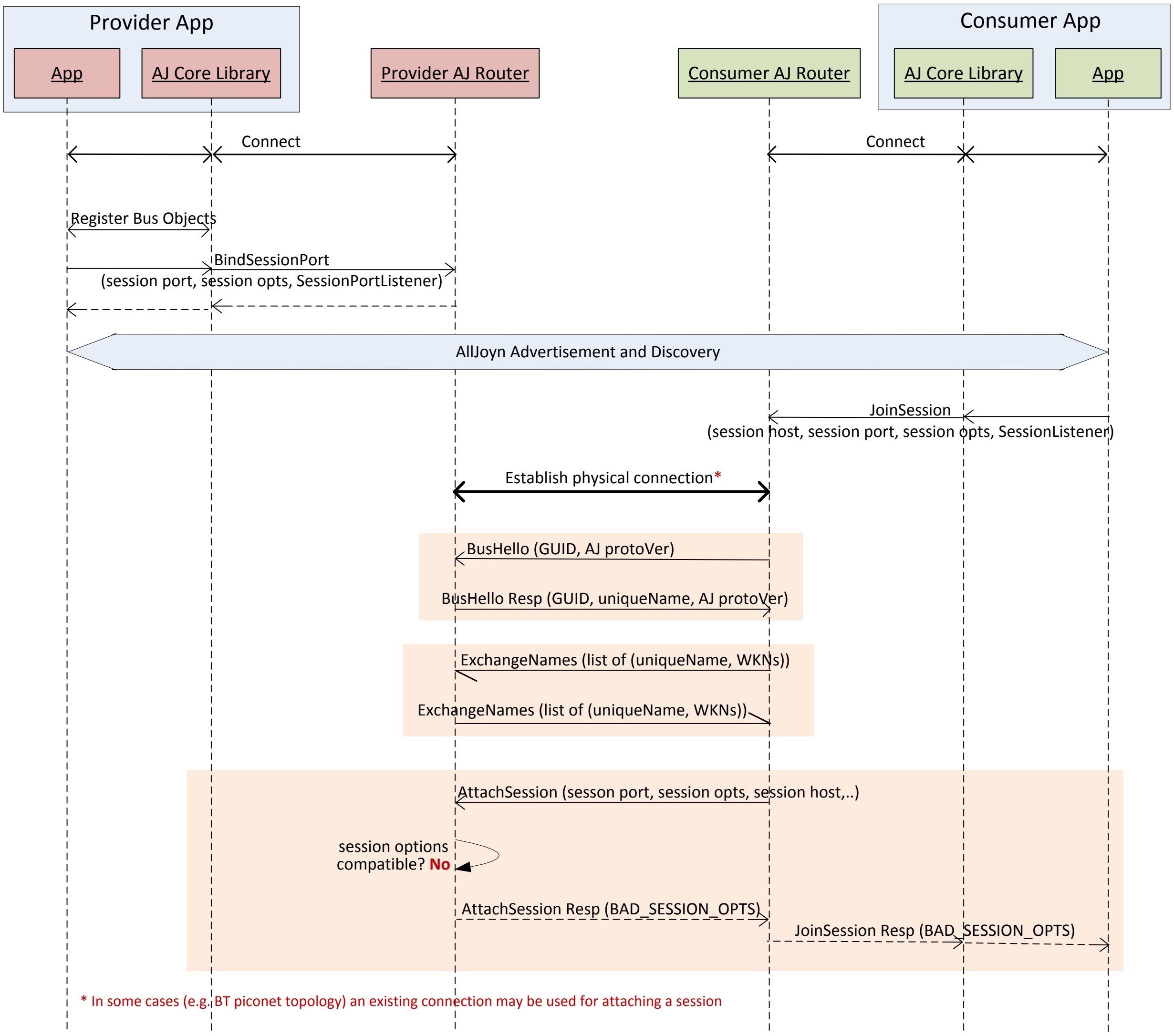

Provider App
App
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer App
AJ Core Library
App
Connect
Connect
Register Bus Objects
BindSessionPort
(session port, session opts, SessionPortListener)
AllJoyn Advertisement and Discovery
JoinSession
(session host, session port, session opts, SessionListener)
Establish physical connection*
BusHello (GUID, AJ protoVer)
BusHello Resp (GUID, uniqueName, AJ protoVer)
ExchangeNames (list of (uniqueName, WKNs))
ExchangeNames (list of (uniqueName, WKNs))
AttachSession (sesson port, session opts, session host,..)
session options
compatible? No
AttachSession Resp (BAD_SESSION_OPTS)
JoinSession Resp (BAD_SESSION_OPTS)
* In some cases (e.g. BT piconet topology) an existing connection may be used for attaching a session

Figure 5-9. AllJoyn session – incompatible session options

5.4 Session options

Table 5-1 captures the session options supported for the AllJoyn session. Traffic, proximity, and transports fields in the session option are specified as bit masks with values.

Table 5-1. Session options

Session option	Description	Data type	Allowed values		
			Name	**Value**	**Description**
traffic	Specifies type of traffic sent over the session	byte	TRAFFIC_MESSAGES	0x01	Use reliable message-based communication to move data between session endpoints.
			TRAFFIC_RAW_UNRELIABLE	0x02	Use unreliable (e.g., UDP) socket-based communication to move data between session endpoints. This creates a raw session where MESSAGE encapsulation is not used.
			TRAFFIC_RAW_RELIABLE	0x04	Use reliable (e.g., TCP) socket-based communication to move data between session endpoints. RAW. This creates a raw session where MESSAGE encapsulation is not used.
isMultipoint	Specifies whether the session is multi-point or point-to-point.	bool	N/A	true	A multi-point capable session. A multi-point session can be joined multiple times to form a single session with multiple (> 2) endpoints.
			N/A	false	Session is not multi-point capable. Each join attempt will create a new point-to-point session.
proximity [4]	Specifies the proximity scope for this session	byte	PROXIMITY_ANY	0xFF	Spatial scope of the session is not limited. Session can be joined by joiners located anywhere.
			PROXIMITY_PHYSICAL	0x01	Spatial scope of session is limited to the local host. Interpreted as "the same physical machine." Session can be joined by joiners located only on the same physical machine as the one hosting the session.
			PROXIMITY_NETWORK	0x02	Spatial scope of session is limited to anywhere on the local logical network segment. Session can be joined by joiners located anywhere on the network.

[4] The PROXIMITY_PHYSICAL and PROXIMITY_NETWORK options are not supported semantically today meaning no enforcement is done for spatial scope. Only bit matching is done for these options when looking to find a set of compatible set of session options. AllJoyn system provides flexibility to support specific semantics for these options in future if needed.

Session option	Description	Data type	Allowed values		
			Name	**Value**	**Description**
transports	Specifies the allowed transports for this Session	short	TRANSPORT_ANY	0xFFFF ~TRANSPORT_WFD	Use any available transport to communicate with a given session except Wi-Fi Direct. The TRANSPORT_WFD bit needs to be explicitly set by the app for use of Wi-Fi Direct transport.
			TRANSPORT_NONE	0x0000	Use no transport to communicate with a given session.
			TRANSPORT_LOCAL	0x0001	Use only the local transport to communicate with a given session.
			TRANSPORT_BLUETOOTH	0x0002	Use only Bluetooth transport to communicate with a given session.
			TRANSPORT_WLAN	0x0004	Use only a wireless local area network to communicate with a given session.
			TRANSPORT_WWAN	0x0008	Use only a wireless wide area network to communicate with a given session.
			TRANSPORT_LAN	0x0010	Use only a wired local area network to communicate with a given session.
			TRANSPORT_ICE	0x0020	Use only ICE transport to communicate with a given session
			TRANSPORT_WFD	0x0080	Use only the Wi-Fi Direct transport to communicate with a given session.

5.4.1 Session options negotiation

A compatible set of session options must be agreed upon between two endpoints to establish a session. If a compatible set of session options cannot be established between two endpoints, session establishment fails.

Session options negotiation occurs between session options provided by the provider app at the time of invoking BindSessionPort(…) and the session options requested by the consumer app when invoking the JoinSession(…).

- For certain session options, e.g., isMultipoint and traffic, exact match must occur between the provider and consumer session options for negotiation to be successful.
- For other session options, the negotiation happens to the lowest common session option level. Exact details of session options negotiation is outside the scope of this document.

5.5 Methods/signals used for an AllJoyn session

The AllJoyn framework supports session-related functionality as part of the following AllJoyn interfaces:

- org.alljoyn.Daemon
- org.alljoyn.Bus
- org.alljony.Bus.Peer.Session

This section provides a summary of methods and signals from these interfaces used for AllJoyn session-related functionality.

5.5.1 org.alljoyn.Daemon

The org.alljoyn.Daemon interface is the main over-the-wire interface used for communication between two AllJoyn router components. Table 5-2 and Table 5-3 summarize the org.alljoyn.Daemon interface methods and signals used for session-related functions.

Table 5-2. org.alljoyn.Daemon interface methods

Method name	Parameters			Description
	Parameter name	Drection	Description	
AttachSession	session port	in	AllJoyn session port	Method for a remote AllJoyn router to attach a session with this AllJoyn router.
	Joiner	in	Unique name of the joiner	
	creator	in	Unique name or well-known name of the session host	
	dest	in	Unique name of the destination for the AttachSession. ■ For point-to-point session, this is same as creator. ■ For multi-point session, this field can be different than the creator.	
	b2b	in	Unique name of the bus-to-bus end point on the joiner side. This is used to set up the message routing path for the session.	
	busAddr	in	A string indicating how to connect to the bus endpoint, for example, “tcp:192.23.5.6, port=2345”	

Method name	Parameters			Description
	Parameter name	Drection	Description	
	optsIn	in	Session options requested by the joiner	
	status	out	Session join status	
	sessionId	out	Assigned session ID	
	optsOut	out	Final selected session options	
	members	out	List of session members	
GetSessionInfo	creator	in	Unique name for the app that bound the session port	Method for a remote AllJoyn router to get session information from this AllJoyn router.
	session port	in	The session port	
	optsIn	in	Session options requested by the joiner	
	busAddr	out	Returned bus address for the session to use when attempting to create a connection for joining the session, for example, “tcp:192.23.5.6, port=2345”	

Table 5-3. org.alljoyn.Daemon interface signals

Signal name	Parameters		Description
	Parameter name	Description	
ExchangeNames	uniqueName	List of one or more unique names available on the local AllJoyn router	A signal that informs remote AllJoyn router of names available on the local AllJoyn router.
	WKNs	List of one or more well-known names registered with each of the known unique name on the local AllJoyn router	
DetachSession	sessionId	AllJoyn session ID	A signal sent out to detach a joiner from an existing session
	Joiner	Unique name of the joiner	

5.5.2 org.alljoyn.Bus

The org.alljoyn.Bus interface is the main AllJoyn interface between the application and the AllJoyn router. Table 5-4and Table 5-5 summarize the org.alljoyn.Bus interface methods and signals used for session-related functions.

Table 5-4. org.alljoyn.Bus interface methods

Method name	Parameters			Description
	Parameter name	Direction	Description	
BusHello	GUIDC	in	GUID of the client AllJoyn router	Method used to exchange identifiers. This can be used between app and AllJoyn router, as well as between two AllJoyn router components.
	protoVerC	in	AllJoyn protocol version of client AllJoyn router	
	GUIDS	out	GUID of the service side AllJoyn router	
	uniqueName	out	Unique name assigned to the bus-to-bus endpoint between two AllJoyn router components.	
	protoVerS	out	AllJoyn protocol version of service side of AllJoyn router	
BindSessionPort	sessionPort	in	Specified session port. Set to SESSION_PORT_ ANY if app is asking AllJoyn router to assign a session port.	Method for an application to initiate binding a session port with the AllJoyn bus.
	opts	in	Specified session options	
	resultCode	out	Result status	
	sessionPort	out	Same as input sessionPort unless SESSION_PORT_ ANY was specified. In the latter case, set to an AllJoyn router-assigned session port.	
UnbindSessionPort	sessionPort	in	Specified session port	Method for an application to unbind a session port with the AllJoyn bus.
	resultCode	out	Result status	

Method name	Parameters			Description
	Parameter name	Direction	Description	
JoinSession	sessionHost	in	Well-known name/unique name of the session creator	Method for an application to initiate joining a session
	sessionPort	in	Specified session port	
	optsIn		Session options requested by the joiner	
	resultCode	out	Result status	
	sessionId	out	Assigned session ID	
	opts	out	Final selected session options	
LeaveSession	sessionId	in	Session ID of the session	Method for an application to initiate leaving an existing session
	resultCode	out	Result status	

Table 5-5. org.alljoyn.Bus interface signals

Signal name	Parameters		Description
	Parameter name	Description	
SessionLost	sessionId	Session ID of the session that was just lost	A signal that informs application when a session ends.
MPSessionChanged	sessionId	Session ID that changed	A signal that informs application on changes to an existing session.
	name	Unique name of the session member that changed	
	isAdd	Flag indicating whether member was added. Set to true if the member has been added.	

5.5.3 org.alljoyn.Bus.Peer.Session

The org.alljoyn.Bus.Peer.Session interface is an AllJoyn interface between application and the AllJoyn router. Table 5-6 and Table 5-7 summarize the org.alljoyn.Bus.Peer.Session interface methods and signals used for session-related functions.

Table 5-6. org.alljoyn.Bus.Peer.Session interface methods

Method name	Parameters			Description
	Parameter name	Direction	Description	
AcceptSession	sessionPort	in	Session port that received the join request	Method for invoking accepting a session locally on the session host
	sessionId	in	ID for the new session (if accepted)	
	creatorName	in	Session creator unique name	
	joinerName	in	Session joiner unique name	
	opts	in	Session options requested by the joiner	
	isAccepted	out	Set ti true if the creator accepts the session	

Table 5-7. org.alljoyn.Bus.Peer.Session interface signals

Signal name	Parameters		Description
	Parameter name	Description	
SessionJoined	sessionPort	Session port of the session which was just lost.	A signal sent locally on the session host to inform it that a session was successfully joined.
	sessionId	ID for the new session	
	creatorName	Session creator unique name	
	joinerName	Session joiner unique name	

6 Data Exchange

6.1 Overview

The AllJoyn provider application implements one or more service objects that provide service functionality. These service objects implement one or more Bus interfaces which support methods, signals, and/or properties as interface members. AllJoyn applications can exchange data using these interface members. An AllJoyn session must be established to exchange data between provider and consumer applications except when sending sessionless signals (see chapter 7).

NOTE

AllJoyn service objects are not tied to any specific AllJoyn session. Any service object can be accessed over any AllJoyn session.

Figure 6-1 shows the functional architecture for the provider application. Aspects about the architecture follow.

- Each service object has an associated object path which app may decide to announce as part of About feature's announcement signal.
- A provider device can host one or more AllJoyn sessions.
- The provider app maintains distinct session port and a session Id for each hosted session among other parameters.
- The provider application endpoint connected to the AllJoyn router has one associated unique name and one or more well-known names advertised by the application.
- The AllJoyn router maintains session related state information. This information is used to perform sessionId-based routing for AllJoyn messages.

Once a session is established, the provider application can communicate with consumer apps via interface methods and properties when invoked by the consumer side. The provider app can also send signals specified in the bus interfaces to send data to consumer apps.

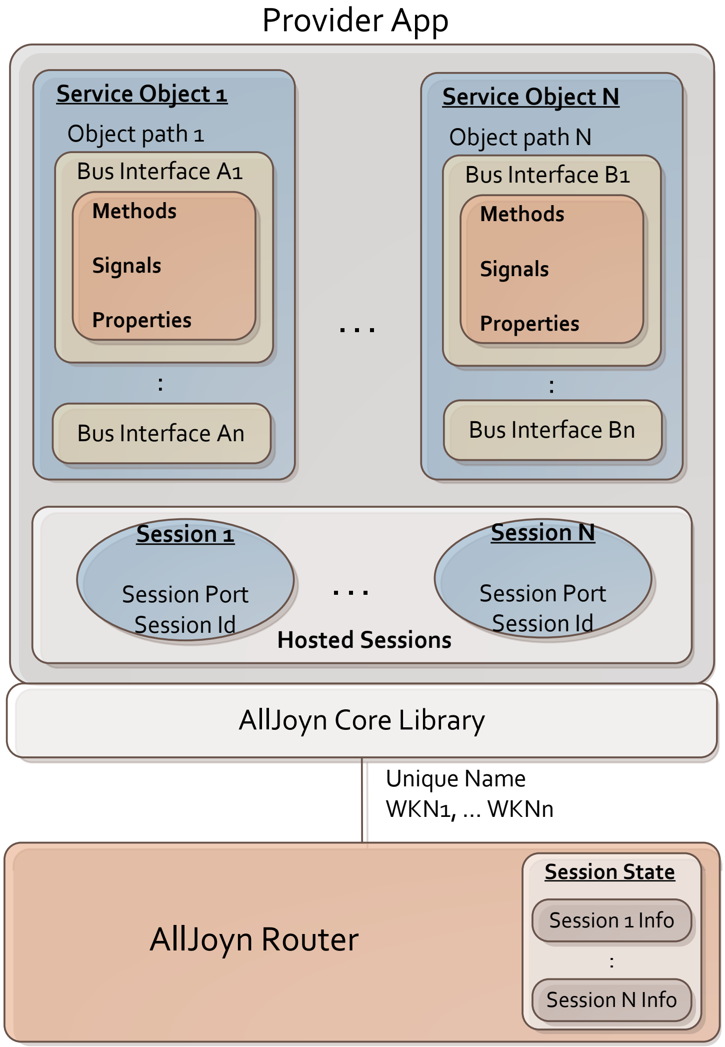


Figure 6-1. Provider functional architecture

After an AllJoyn session is established, the consumer application has the connection established to exchange data with the provider app. The consumer app in a session can invoke methods and properties on the remote service objects, or can opt to receive signals emitted by the provider app. A ProxyBusObject is needed to exchange data via methods and properties. A signal handler is needed to receive signal data from the provider app.

Figure 6-2 shows the functional architecture for the consumer application. Aspects of this architecture follow.

- A consumer app can join one or more AllJoyn sessions which can be with the same or different provider apps.
- The consumer app creates one or more proxy objects, one for each remote service object it wants to communicate with.
  - A proxy object is a local representation of the desired remote service object on the provider app.

  - ❒ The service object path, unique name of the provider app and sessionId information are provided to create a proxy object.

- To receive signals from the provider app, consumer app registers specific signal handlers with the AllJoyn router for signal names specified in the service object.
- When a particular signal is received, specified signal handler gets invoked.
- The consumer application endpoint connected to the AllJoyn router has one associated unique name.
- The AllJoyn router maintains session-related state information for the joined sessions. This information is used to perform sessionId based routing for AllJoyn messages.

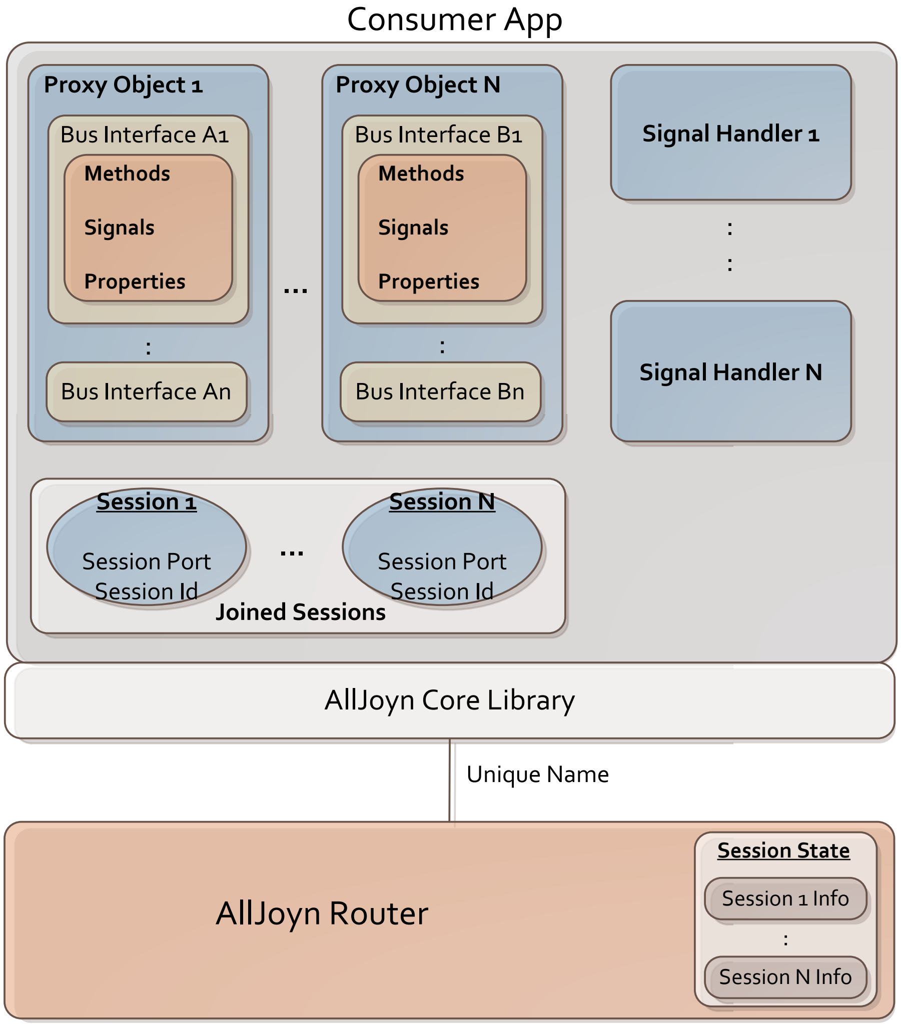


Figure 6-2. Consumer functional architecture

6.2 Data exchange via methods

The following use cases illustrate data exchange via method calls:

- Provider app sending a reply
- Provider app not sending a reply

6.2.1 Provider app sending a reply

Figure 6-3 captures the message flow for the scenario when a consumer app remotely invokes a method call on the provider app to exchange data. A METHOD_RETURN reply message is sent back to the consumer app.

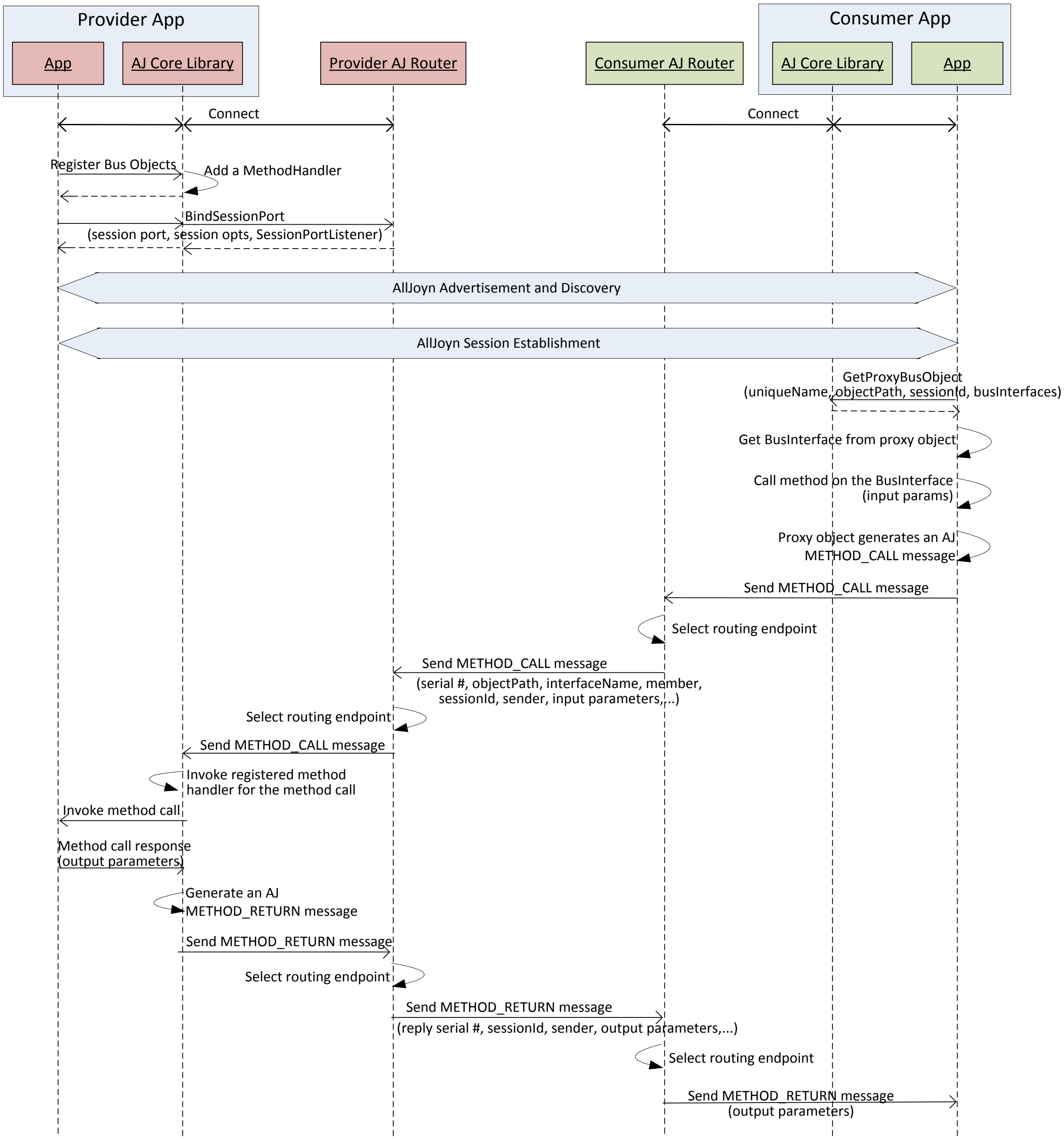


Figure 6-3. Data exchange via method calls (reply sent)

The message flow steps are described below.

1. Both the provider and consumer apps connect to the AllJoyn router and perform advertisement and discovery steps to discover desired services.
2. The provider app registers its service objects with the AllJoyn core library. This step is needed to expose service objects to remote nodes on the network. The AllJoyn

core library adds a MethodHandler for all the methods associated with a given service object.

3. The provider app binds a session port with the AllJoyn router via the AllJoyn core library's BindSessionPort API. This call specifies a session port, session options and a SessionPortListener for the session.
4. The provider and consumer apps perform AllJoyn service advertisement and discovery to discover the service offered by the provider app.
5. The consumer app establishes an AllJoyn session with the provider app over the bound session port. Now there is a session connection established between provider and consumer app to exchange data.
6. The consumer app creates a proxyBusObect via the AllJoyn core library's GetProxyBusObjcet API. The app specifies unique name of provider app, object path for the service object, sessionId, and list of BusInterfaces to which the proxy object should respond.
7. The consumer app gets the BusInterface from the created proxy object and calls a method on this BusInterface. App provides input parameters for the method.
8. The ProxyBusObject:MethodCall method is called, which generates an AllJoyn METHOD_CALL message for the method call.
9. The proxy object sends the generated METHOD_CALL message to the AllJoyn router.
10. The AllJoyn router receives the message and determines where this message needs to be routed based on the session ID/destination information included in the message. In this case, the message needs to be routed to remote AllJoyn router endpoint at the provider.
11. The AllJoyn router sends the METHOD_CALL message to the remote AllJoyn router over the established session connection. The METHOD_CALL message includes a serial number, service objectPath, interfaceName in the service object, member name (method name) within the interfaceName, sessionId, and sender unique name as part of the header fields of the message. The method input parameters are included as part of the message body.
12. The provider AllJoyn router receives the METHOD_CALL message. It determines where this message needs to be routed based on the session ID/destination information. In this case, the message needs to be routed to the AllJoyn core library app endpoint.
13. The AllJoyn router sends the METHOD_CALL message to the AllJoyn core library endpoint.
14. The AllJoyn core library invokes the registered MethodHandler for the method call member specified in the received message. The MethodHandler invokes the actual method call with the BusInterface of the service object and receives the method response. It generates a METHOD_RETURN message for the method reply and sends to the AllJoyn router.
15. The AllJoyn router receives the METHOD_RETURN message and determines where this message needs to be routed based on session ID/destination information

included in the message. In this case, the message needs to be routed to the remote AllJoyn router endpoint at the consumer.

- The provider AllJoyn router sends the METHOD_RETURN message to the remote AllJoyn router over established session connection. The METHOD_RETURN message includes a reply serial number (serial number of the associated METHOD_CALL message), sessionId, and sender's unique name as part of message header fields. Any output parameters for the METHOD_RETURN message is specified as part of the message body.
- The consumer AllJoyn router receives the METHOD_RETURN message. It determines where this message needs to be routed based on the session ID/destination information. In this case, the message needs to be routed to the app endpoint.
- The AllJoyn router sends the METHOD_RETURN message to the application's endpoint with output parameters as the response to the original METHOD_CALL message. If the METHOD_CALL message was sent asynchronously, a ReplyHandler gets registered which is invoked when the METHOD_RETURN message is received.

6.2.2 Provider app not sending a reply

When defining an interface definition, the provider can annotate some methods that are not returning any output parameters as NO_REPLY_EXPECTED. For such methods, the provider app will not be sending any METHOD_RETURN message back to the consumer. When invoking such a method, the consumer should set the NO_REPLY_EXPECTED flag to indicate to the AllJoyn core library that it does not need to start a timer and wait for a reply from the provider.

Figure 6-4 shows the message flow for the scenario when a method is invoked with the NO_REPLY_EXPECTED flag set, and no reply is sent by the provider app.

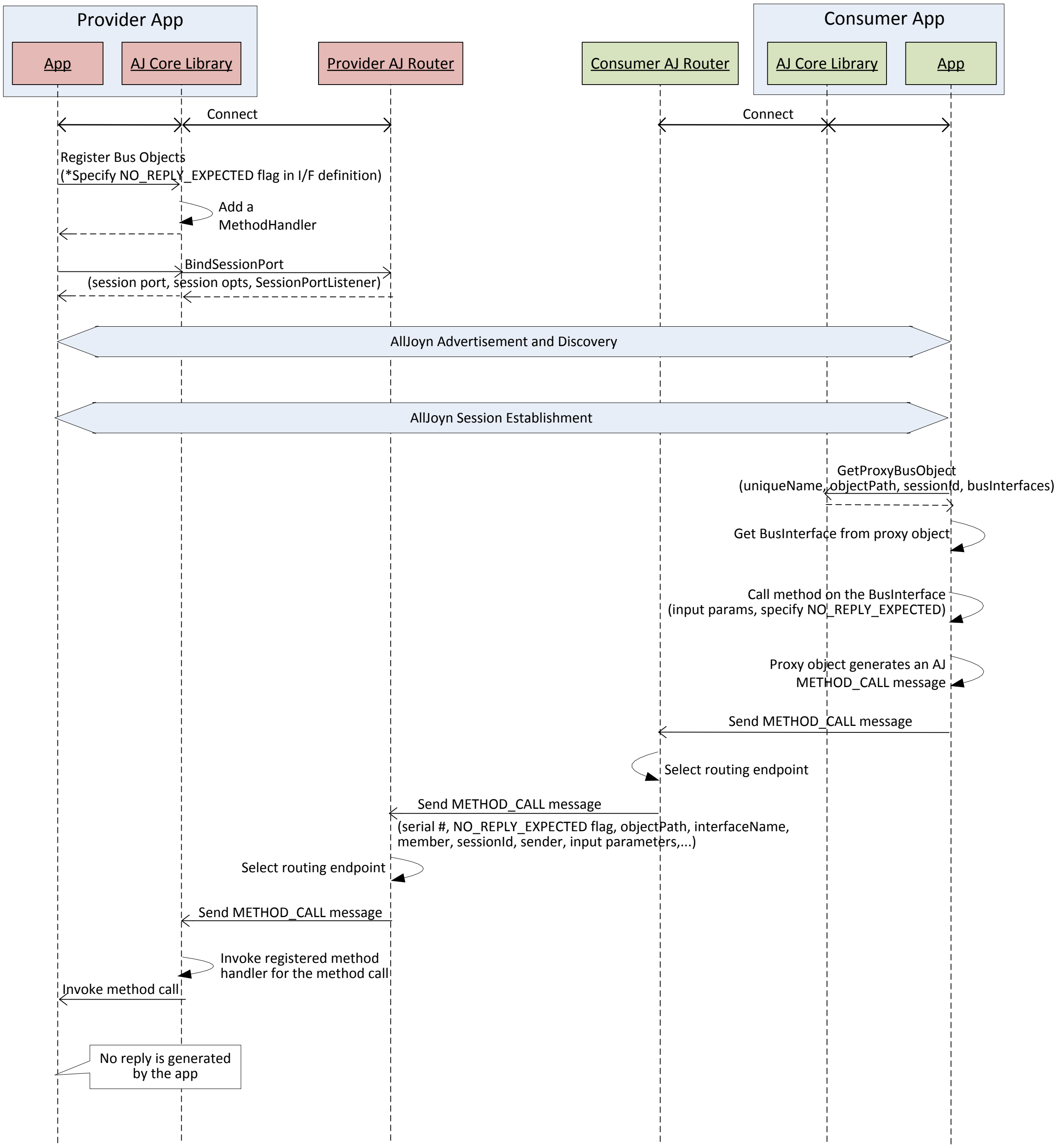

Provider App
App
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer App
AJ Core Library
App
Connect
Connect
Register Bus Objects
(*Specify NO_REPLY_EXPECTED flag in I/F definition)
Add a
MethodHandler
BindSessionPort
(session port, session opts, SessionPortListener)
AllJoyn Advertisement and Discovery
AllJoyn Session Establishment
GetProxyBusObject
(uniqueName, objectPath, sessionId, busInterfaces)
Get BusInterface from proxy object
Call method on the BusInterface
(input params, specify NO_REPLY_EXPECTED)
Proxy object generates an AJ
METHOD_CALL message
Send METHOD_CALL message
Select routing endpoint
Send METHOD_CALL message
(serial #, NO_REPLY_EXPECTED flag, objectPath, interfaceName,
member, sessionId, sender, input parameters,...)
Select routing endpoint
Send METHOD_CALL message
Invoke registered method
handler for the method call
Invoke method call
No reply is generated
by the app

Figure 6-4. Data exchange via method calls (reply not sent)

Most of the message flow steps are similar to the message flow in Figure 6-3 returning a method reply. The following captures the distinguishing steps with respect to that flow.

1. When defining a service interface, provider app annotates one or more methods in the interface as NO_REPLY_EXPECTED. The service object implementing the service interface gets registered with the AllJoyn core library.

2. On the consumer side, when that method call is invoked via ProxyObject interfaces, the consumer sets the NO_REPLY_EXPECTED flag. This indicates to the AllJoyn core library that it does not need to start a timer and wait for a reply from the provider.
3. The METHOD_CALL message reaches the provider app and the associated method gets invoked. Since the method was annotated with NO_REPLY_EXPECTED, the provider app does not generate any reply.

6.3 Data exchange via signals

Figure 6-5 captures the message flow for the scenario when consumer app registers to receive signals from the provider app to exchange data.

A signal gets forwarded to all participants in a session if only sessionId is specified in the header (no destination field).

If a specific destination is specified in the header field, then the signal is only sent to that participant.

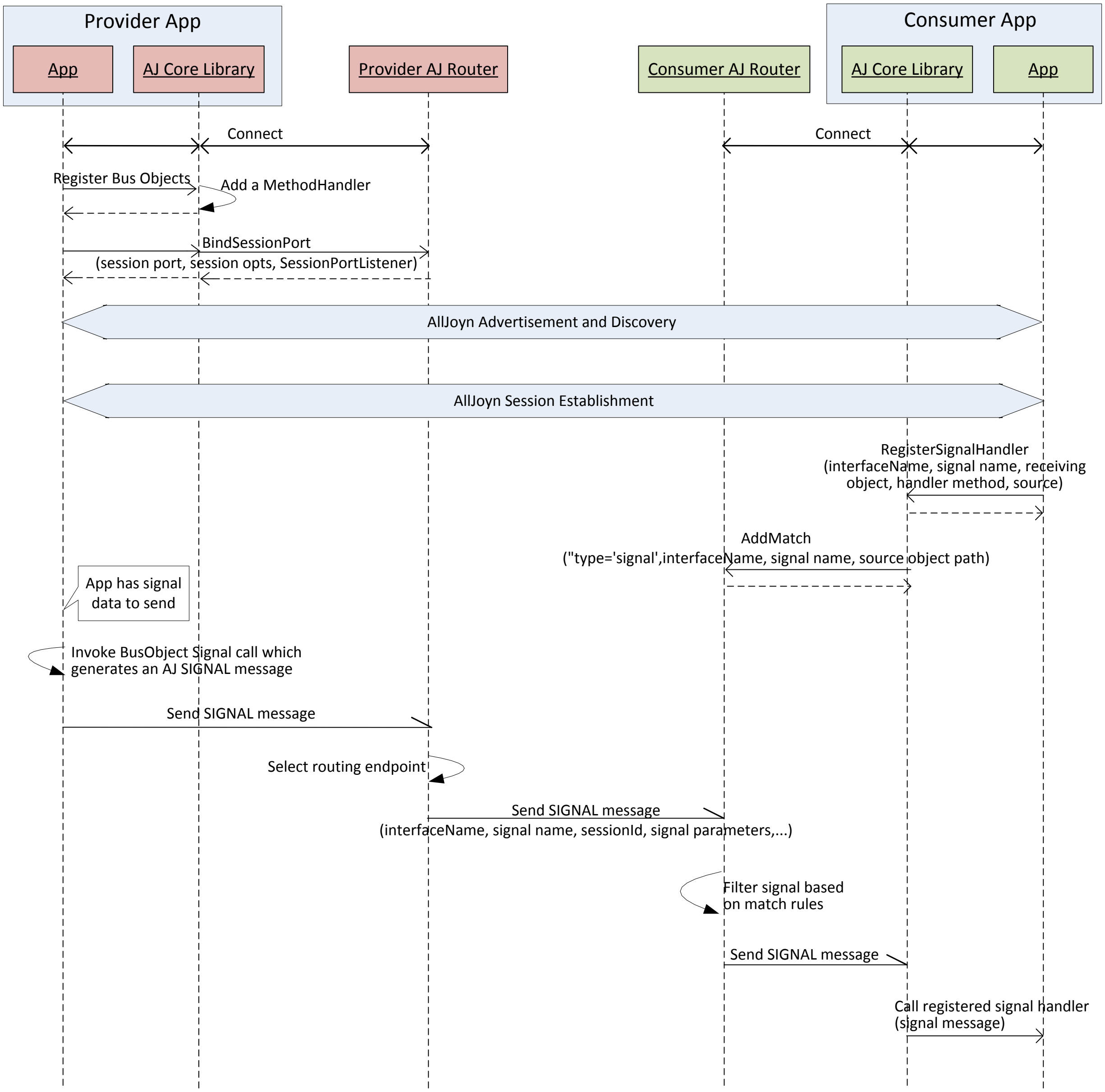

Provider App
App
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer App
AJ Core Library
App
Connect
Connect
Register Bus Objects
Add a MethodHandler
BindSessionPort
(session port, session opts, SessionPortListener)
AllJoyn Advertisement and Discovery
AllJoyn Session Establishment
RegisterSignalHandler
(interfaceName, signal name, receiving object, handler method, source)
AddMatch
("type='signal',interfaceName, signal name, source object path)
App has signal data to send
Invoke BusObject Signal call which generates an AJ SIGNAL message
Send SIGNAL message
Select routing endpoint
Send SIGNAL message
(interfaceName, signal name, sessionId, signal parameters,...)
Filter signal based on match rules
Send SIGNAL message
Call registered signal handler
(signal message)

Figure 6-5. Data exchange via signals

The message flow steps are described below.

1. Both the provider and consumer apps connect to the AllJoyn router and perform advertisement and discovery steps to discover desired services.
2. The provider app registers its service objects with the AllJoyn core library. This step is needed to expose service objects to remote nodes on the network. The AllJoyn core library adds a MethodHandler for all the methods associated with a given service object.

3. The provider app binds a session port with the AllJoyn router via the AllJoyn core library's BindSessionPort API. This call specifies a session port, session options, and a SessionPortListener for the session.
4. The provider and consumer apps perform AllJoyn service advertisement and discovery to discover the service offered by the provider app.
5. The consumer app establishes an AllJoyn session with the provider app over the bound session port. Now there is a session connection established between provider and consumer app to exchange data.
6. The consumer app registers a signal handler for receiving specific signal from the provider side service object by invoking RegisterSignalHandler API with the AllJoyn core library. The app specifies the interface name for the interface which includes the signal, name of the signal, object path for the object receiving the signal, signal handler method, and source object path for signal emitter.
7. The AllJoyn core library invokes an AddMatch call with the AllJoyn router to register the rule to receive this signal. This rule specifies type=signal, interface name, signal member name, and source object path for the object generating the signal.
8. When the provider app has the signal to send, it invokes the BusObject Signal(…) call specifying all the signal parameters. This call generates an AllJoyn SIGNAL message.
9. The SIGNAL message gets sent to the AllJoyn router.
10. The AllJoyn router receives the SIGNAL message and determines where this message needs to be routed based on the session ID/destination information included in the message. In this case, the message needs to be routed to remote AllJoyn router endpoint at the consumer.
11. The consumer AllJoyn router receives the signal and filters it based on registered match rules. In this case, the signal matches the registered rule. The AllJoyn router sends the received SIGNAL message to the AllJoyn core library app endpoint.
12. The AllJoyn core library calls the registered signal handler for the signal passing the received signal message parameters.

6.4 Data exchange via property

The provider and consumer apps can exchange data via property members defined in the BusInterfaces of the service object. A property member has predefined get and set method calls to get the value of the property and to set a specific value on the property. The consumer app can invoke these get and set method calls for the property to exchange data.

To invoke these property methods, the message flow detailed for method calls in section 6.2 applies.

6.5 Signal versus (method call without reply)

It is good to understand the difference between a signal and method call without a reply. In both cases, a single message is sent from the source; however, these are quite different. One of the main differences is that they are sent in different directions. A SIGNAL message is emitted by the provider app whereas a METHOD_CALL message is sent by the consumer app. In addition, a SIGNAL message can be sent to be received by either a single destination or by multiple destinations, whereas the METHOD_CALL message is always sent to a single destination.

6.6 Match rules

The AllJoyn framework supports D-Bus match rules for the consumer application to request and receive specific set of messages. Match rules describe the messages that should be sent to a consumer app based on the contents of the message. Match rules are typically used to receive a specific set of signal messages. Consumer apps can ask to receive specific set of signals from the AllJoyn router by specifying filtering/matching rules for signals.

Signals that are sent to a specific destination do not need to match the consumer’s match rule. Instead, match rules are applied to signals which are not sent to a specific destination; they are meant to be received by multiple endpoints. These include broadcast signals, sessionless signals and session-specific signals sent to multiple participants in the session. Such signals get forwarded to only those consumer applications that have a suitable match rule. This avoids unnecessary waking up and processing for signals at the consumer applications.

Consumer applications can add a match rule by using the AddMatch method exposed by the AllJoyn router. Match rules are specified as a string of comma-separated key/value pairs. Excluding a key from the rule indicates a wildcard match, e.g., excluding the member key from a match rule but adding a sender lets all messages from that sender through.

For example:

> Match Rule = "type='signal',sender='org.freedesktop.DBus',interface='org.freedesktop.DBus',member='Foo',path='/bar/foo',destination=':452345.34'"

The AllJoyn framework supports a subset of D-Bus match rules as captured in Table 6-1.

NOTE

The AllJoyn does not support D-Bus specified arg[0,1…N], arg[0,1,…N]path, arg0namespace and eavesdrop=’true’ in match rules.

Table 6-1. Match rule keys supported by the AllJoyn framework

Match key	Possible values	Description
type	■ signal ■ method_call ■ method_return ■ error	Match on the message type. An example of a type match is type='signal'.
sender	A well-known name or unique name	Match messages sent by a particular sender. An example of a sender match is sender='org.alljoyn.Refrigerator'.
interface	An interface name	Match messages sent over or to a particular interface. An example of an interface match is interface='org.alljoyn.Refrigerator'. If a message omits the interface header, it must not match any rule that specifies this key.
member	Any valid method or signal name	Matches messages which have the give method or signal name. An example of a member match is member='NameOwnerChanged'.
path	An object path	Matches messages which are sent from or to the given object. An example of a path match is path='/org/alljoyn/Refrigerator'.
path_namespace	An object path	Matches messages which are sent from or to an object for which the object path is either the given value, or that value followed by one or more path components. For example, path_namespace='/com/example/foo' would match signals sent by /com/example/foo or by /com/example/foo/bar, but not by /com/example/foobar. Using both path and path_namespace in the same match rule is not allowed.
destination	A unique name	Matches messages which are being sent to the given unique name. An example of a destination match is destination=':100.2'.

An application can add multiple match rules for signals with the AllJoyn router. In this case, the app is essentially requesting to get signal messages based on multiple filtering criteria, and all match rules are applicable. As a result, the signal messages get sent to the app if they matches any of the specified match rules.

The AllJoyn router sends a union of messages to the app that matches with the specified rules. For example, if there is a more restrictive rule that matches a small set of signals, and there is another less restrictive rule that matches a larger superset of signals, the AllJoyn router always sends the larger superset of signals to the app.

6.7 Type system

The AllJoyn framework uses the D-Bus protocol type system which allows values of various types to be serialized in a standard way into a sequence of bytes referred to as the wire format. Converting values from some other representation into the wire format is called marshaling, and converting it back from the wire format is called unmarshaling.

The AllJoyn framework uses D-Bus marshaling format.

6.7.1 Type signatures

The AllJoyn framework uses the same type signatures that are used by the D-Bus protocol. The type signature is made up of type codes. The type code is an ASCII character that represents a standard data type.

Table 6-2 lists the data types used by the AllJoyn framework.

Table 6-2. Data types supported by the AllJoyn framework

Conventional name	Code	ASCII	Description
INVALID	0	NUL	Not a valid type code, used to terminate signatures
BYTE	121	‘y’	8-bit unsigned integer
BOOLEAN	98	‘b’	Boolean value, 0 is FALSE and 1 is TRUE. Everything else is invalid.
INT16	110	‘n’	16-bit signed integer
UINT16	113	‘q’	16-bit unsigned integer
INT32	105	‘i’	32-bit signed integer
UINT32	117	‘u’	32-bit unsigned integer
UINT64	120	‘x’	64-bit signed integer
DOUBLE	100	‘d’	IEEE 754 double
STRING	115	‘s’	UTF-8 string (must be valid UTF-8). Must be null terminated and contain no other null bytes.
OBJECT_PATH	111	‘o’	Name of an object instance
SIGNATURE	103	‘g’	A type signature
ARRAY	97	‘a’	Array
STRUCT	114, 40, 41	‘r’, ‘(‘, ‘)’	Struct
VARIANT	118	‘v’	Variant type (the type of the value is part of the value itself)
DICT_ENTRY	101, 123, 125	'e','{','}'	Entry in a dict or map (array of key-value pairs)

Four of the types are container types: STRUCT, ARRAY, VARIANT, and DICT_ENTRY. All other types are common basic data types. When specifying a STRUCT or DICT_ENTRY, 'r' and 'e' should not be used. Instead, ASCII characters '(', ')', '{', and '}' should be used to mark the beginning and ending of a container.

6.8 Message format

The AllJoyn framework uses the D-Bus message format and extends it with additional header flags and header fields for AllJoyn messages. The AllJoyn message format is used to send messages between AllJoyn routers as well as between the application and the AllJoyn router.

Method calls, method replies and signal messages get encapsulated in AllJoyn message format. D-Bus defined METHOD_CALL, METHOD_RETURN and SIGNAL messages are used (with AllJoyn enhancements) for transporting these messages respectively. In case of error scenarios, an ERROR message is returned in reply to a method call (instead of METHOD_RETURN).

An AllJoyn message consists of a header and a body. Figure 6-6 shows the AllJoyn message format. Definitions for each message format field are provided in subsequent tables.

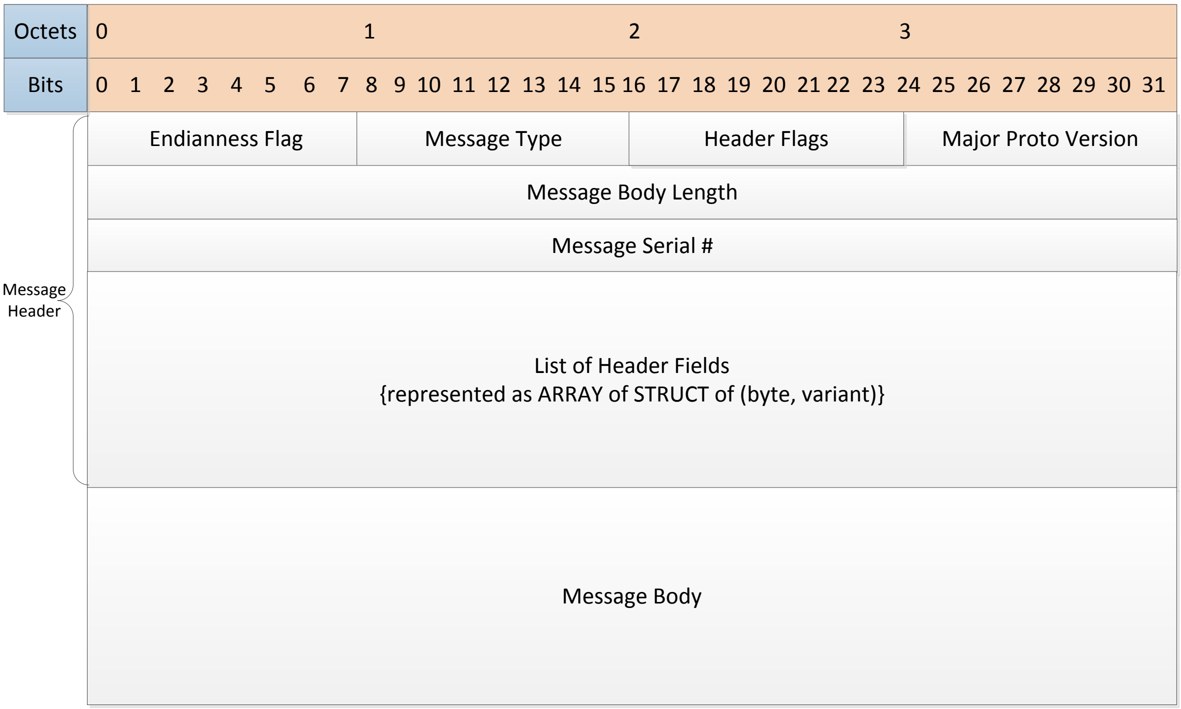


Figure 6-6. AllJoyn message format

Table 6-3. Message format fields supported by the AllJoyn framework

Field name	Description
Endianness Flag	Endianness of the message. ASCII 'l' for little-endian or ASCII 'B' for big-endian. Both header and body are in this endianness.
Message Type	Type of message. This field is set per the definitions specified in Table 6-4.
Header Flags	Provides any applicable flags for the message. This field is bitwise OR of flags. Unknown flags must be ignored. This is set per the definitions specified in Table 6-5.
Major Protocol Version	AllJoyn major protocol version for the sending application of this message.
Message Body Length	Length (in bytes) of the message body, starting from the end of the header.
Serial Number	Serial number of this message. This is assigned by the sender and used as a cookie by the sender to identify the reply corresponding to this request. This must not be zero.

Field name	Description
List of Header Fields	This specifies an array of zero or more header fields where each field is a 1-byte field code followed by a field value. This is represented as ARRAY of STRUCT of (BYTE, VARIANT). A header must contain the required header fields for its message type, and zero or more of any optional header fields. Implementations must ignore fields they do not understand. The AllJoyn framework has extended the list of D-Bus defined header fields. Table 6-6 lists all the header fields supported by AllJoyn and mandatory/optional requirement for these fields for different message types.
Message Body	Body of the message. The content of message body is interpreted based on SIGNATURE header field.

Table 6-4. Message Type definitions

Name	Value	Description
INVALID	0	An invalid type
METHOD_CALL	1	Method call
METHOD_RETURN	2	Method reply with returned data
ERROR	3	Error reply
SIGNAL	4	Signal emission

Table 6-5. Header Flag definitions

Name	Value	Description
NO_REPLY_EXPECTED	0x01	Indicates that no reply (method_return or error) is expected for the Method Call. The reply can be omitted as an optimization. **NOTE** The provider app can still send back a reply despite this flag.
AUTO_START	0x02	Indicates a request to start the service if not running. It is up to the AllJoyn core to honor this or not. **NOTE** This flag is currently not supported.
ALLOW_REMOTE_MSG	0x04	Indicates that messages from remote hosts should be allowed (valid only in Hello message sent from app to the AllJoyn core). If set by the app, the AllJoyn core allows messages from remote apps/hosts to be sent to the application.
(Reserved)	0x08	Reserved/Unused
SESSIONLESS	0x10	Indicates a sessionless signal message
GLOBAL_BROADCAST	0x20	Indicates a global (bus-to-bus) broadcast signal. Applicable for signal only when SESSION_ID=0. If set, the associated signal gets delivered to all the nodes connected over any session in the proximal network.
COMPRESSED	0x40	Indicates that the AllJoyn message header is compressed
ENCRYPTED	0x80	Indicates that the AllJoyn message body is encrypted

Table 6-6. Header Fields definitions

Name	Field code	Type	Required in	Description
INVALID	0	N/A	Not allowed	Not a valid field name (error if it appears in a message)
PATH	1	OBJECT_PATH	METHOD_CALL SIGNAL	Path of the object to send a method call to or path of the object a signal is emitted from.
INTERFACE	2	STRING	METHOD_CALL SIGNAL	Interface to invoke a method call on, or the interface that a signal is emitted from.
MEMBER	3	STRING	METHOD_CALL SIGNAL	The member, either the method name or signal name.
ERROR_NAME	4	STRING	ERROR	Name of the error that occurred, for error messages.
REPLY_SERIAL	5	UINT32	■ ERROR ■ METHOD_RETURN	Serial number of the message this message is a reply to.
DESTINATION	6	STRING	optional for SIGNAL Required for all other messages.	The unique name of the connection this message is intended for.
SENDER	7	STRING	Required in all messages	The unique name of the sending connection. The message bus fills in this field
SIGNATURE	8	SIGNATURE	optional	The data type signature of the message body. This is specified using D-Bus data type system. If omitted, it is assumed to be the empty signature implying that the body must be of 0-length.
N/A	9	N/A	N/A	Unused
TIMESTAMP	10	UINT32	optional	Timestamp when the message was packaged.
TIME_TO_LIVE	11	UINT16	optional If not specified, TTL is assumed to be infinite.	TTL associated with the message. A message gets discarded by the AllJoyn router when the TTL expires. ■ For sessionless signal, the TTL value is specified in seconds. ■ For other messages, the TTL value is specified in msec.

Name	Field code	Type	Required in	Description
COMPRESSION_TOKEN	12	UINT32	optional	Token generated for the messages with header compression on.
SESSION_ID	13	UINT32	optional	Session ID for the session over which this message is being sent. If missing, it is assumed to be 0.

6.9 Message routing

The AllJoyn system supports routing logic to route the following categories of messages:

- App-specific messages: These are app-generated messages that get routed between app endpoints based on the session ID/destination based routing logic described in section 6.9.1.
- Control messages: These messages are generated by the AllJoyn router (e.g., AttachSession) that get routed to the local endpoint of the AllJoyn router.

6.9.1 Session ID/destination-based routing

The AllJoyn system supports message routing based on session ID and/or destination fields for app-specific messages. A session ID-based routing table is formed and maintained at the AllJoyn router for routing messages. A single routing table is maintained for all the active sessions.

Conceptually, for every session ID, the routing table maintains a list of destination app endpoints for every app participating in the session and next hop bus-to-bus endpoint for those app endpoints which are remote. A remote endpoint is attached to a different AllJoyn router; however, it can be on the same device or on a different device. For destination endpoints that are local to the AllJoyn router, no bus-to-bus endpoint is maintained in the routing table.

> AllJoyn routing table = List (session Id, List (destination app endpoint, next hop B2B endpoint))

NOTE

A given destination endpoint can appear multiple times as part of different sessionId entries in an AllJoyn routing table. In this case, if there are multiple possible paths to a remote destination, different bus-to-bus endpoints can be used for the same destination as part of different sessionId entries.

When selecting a route, sessionId is used first to find a matching entry in the routing table. Destination field is used next to select a bus-to-bus endpoint (for remote destinations).

Figure 6-7 shows a deployment with two devices having an AllJoyn session established between them. All four apps are part of the session.

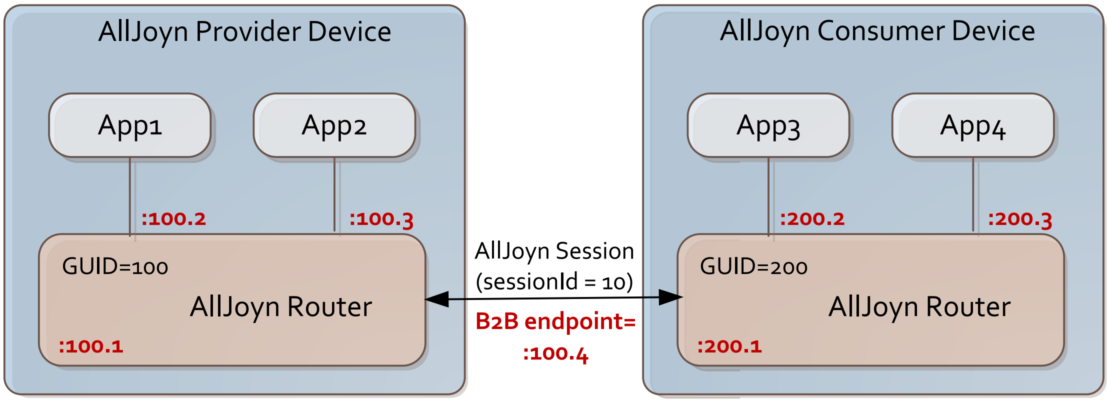


Figure 6-7. AllJoyn routing example

The AllJoyn router on each of the device maintains a routing table. Table 6-7 and Table 6-8 show sample AllJoyn routing tables maintained on the provider and consumer AllJoyn routers, respectively.

Table 6-7. Sample routing table on provider device

Session ID	Destination (App Endpoint)	Next Hop (B2B Endpoint)
10	App1 Endpoint (:100.2)	NA
	App 2 Endpoint (:100.3)	NA
	App 3 Endpoint (:200.2)	B2B Endpoint (:100.4)
	App 4 Endpoint (:200.3)	B2B Endpoint (:100.4)

Table 6-8. Sample routing table on consumer device

Session ID	Destination (App Endpoint)	Next Hop (B2B Endpoint)
10	App1 Endpoint (:100.2)	B2B Endpoint (:100.4)
	App 2 Endpoint (:100.3)	B2B Endpoint (:100.4)
	App 3 Endpoint (:200.2)	NA
	App 4 Endpoint (:200.3)	NA

6.9.1.1 Routing table formation

Routing tables are formed based on the bus-to-bus endpoint information included in the AttachSession method call. When an AllJoyn router receives an AttachSession call, it can be from an app trying to form a new session or from a new member being added to an existing session.

- AttachSession for a new session: In this case, the AllJoyn router sends an Accept session to the app.[5] If the session is accepted, it creates a new sessionId. It then adds an entry for that sessionId in the routing table with the two participants as destinations and bus-to-bus endpoint received in the AttachSession as next hop for the remote app endpoint.
- AttachSession from an added member: In this case, the session is a multi-point session and the AllJoyn router already has an entry for the associated sessionId in the routing table. The member from where the AttachSession is received gets added as a new destination with the bus-to-bus endpoint in the AttachSession as next hop.

6.9.1.2 Routing logic

As described above, the sessionId from the message (if present) is used first to find a matching sessionId entry in the routing table. Next, the destination field (if present) is used to find a matching destination entry to perform the routing.

The following sections capture the routing logic for different use cases.

6.9.1.2.1 Routing based on sessionId and destination fields

If an app-directed message has a non-zero sessionId as well as destination fields, the AllJoyn router first finds that sessionId entry in the routing table and then finds the destination entry within that sessionId for the message destination.

- If the destination was a remote endpoint, then the message gets sent to the bus-to-bus endpoint specified for that destination in the routing table.
- If the destination is locally attached to the AllJoyn router, the message gets directly sent over the local bus connection to that destination.

6.9.1.2.2 Routing based on sessionId field only

If an app-directed message only has a sessionId but no destination field, the message gets forwarded to all the destination endpoints in that session. The AllJoyn router finds the matching sessionId entry in the routing table and send the message to all the destinations listed for that session, except the one which sent the message.

- For remote app endpoints in the session, the message gets forwarded to associated bus-to-bus endpoint from the routing table.
- For locally attached app endpoints in the session, the AllJoyn router directly forwards message to those app endpoints over the local bus connection.

[5] Currently, the single-hop use case is captured.

6.9.1.2.3 Routing for sessionId=0

An app-directed message can specify a sessionId=0 or, if no sessionId field is included, the AllJoyn router assumes sessionId to be 0. The sessionId field value can be zero for any message type. For METHOD_CALL, METHOD_RETURN and ERROR messages, the only requirement is that the destination field must be specified.

For messages with sessionId=0 (or no specified sessionId), if a destination field is specified, the AllJoyn router selects any available route from the routing table (from any of the session entry containing that destination) and forwards the message over the bus-to-bus endpoint for that route.

For SIGNAL messages with sessionId=0 (or no specified sessionId), the destination field does not need to be present. In this case, the AllJoyn router looks at the GLOBAL_BROADCAST flag in the message to determine how that SIGNAL message should be routed per logic below:

- GLOBAL_BROADCAST Flag set: The SIGNAL message should be globally broadcast to all connected endpoints over any session. The Destination field is not looked at when routing such a signal message. The AllJoyn router sends this message to all destination endpoints from the routing table across all sessionIds.
    - For remote destinations, the SIGNAL message gets forwarded to the associated bus-to-bus endpoint.
    - For locally connected destination, the message gets forwarded directly to the app endpoint over local bus connection.
- GLOBAL_BROADCAST Flag not set: The SIGNAL message should be sent over all the locally attached app endpoints. The AllJoyn router forwards the message to all of the locally connected app endpoint over local bus connection.

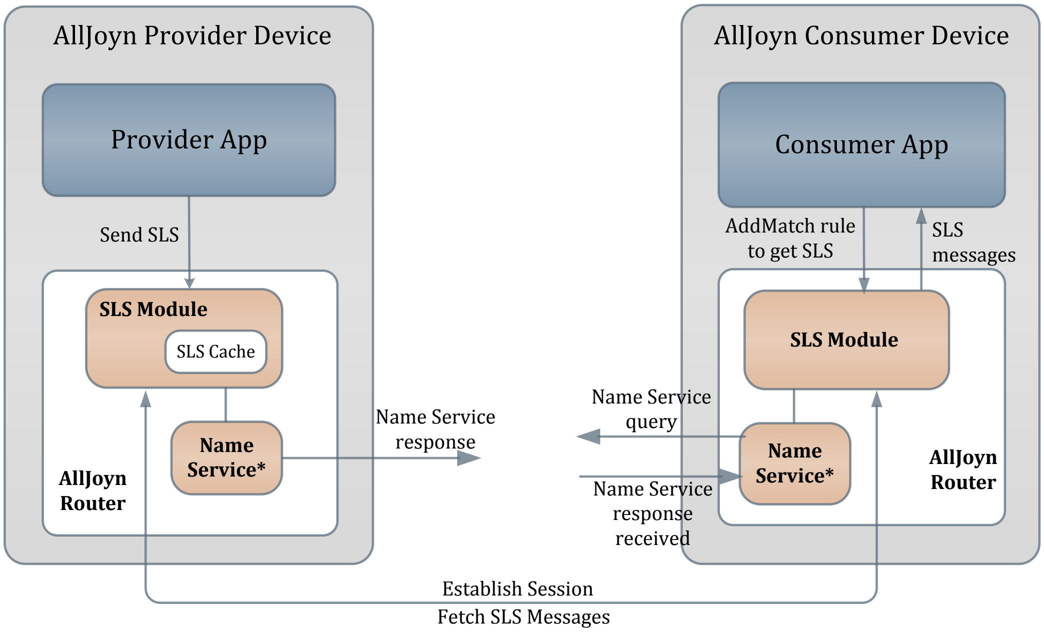
7 Sessionless Signal

7.1 Overview

The sessionless signal is an AllJoyn feature that enables broadcasting of signals to all reachable nodes in the AllJoyn proximal network. This is different than the session-based signals described in Chapter 6, where signals are sent only to participants connected over a given session or multiple sessions (for session broadcast signals) based on sessionId/destination based routing.

Sessionless signals are logically broadcast signals and any app on the AllJoyn proximal network interested in receiving sessionless signals will receive all sessionless signals sent by any other app on that network. The AllJoyn system design refers to sessionless signals as logically broadcast because signals themselves are not broadcast/multicast, only an indication for signals is sent over multicast to all the nodes on the network. Applications do not have to be connected over sessions to receive sessionless signals, however, the AllJoyn router underneath must establish a session to fetch these signals based on the indication received. Applications can specify match rules (via AddMatch) to receive a specific set of sessionless signals and the AllJoyn router filters out signals based on those match rules.

Figure 7-1 captures the high-level architecture for the sessionless signal on the provider and consumer sides. The AllJoyn router supports a logical SLS module that implements sessionless signal logic. The SLS module makes use of the Name Service to advertise and discover sessionless signals using a sessionless signal-specific well-known name.



*Name Service can be legacy Name Service or NGNS depending upon the AllJoyn router release version.

Figure 7-1. Sessionless signal architecture

After AllJoyn router startup, the SLS module executes the following steps to prepare itself for sending and/or receiving sessionless signals.

1. Create an object implementing “org.alljoyn.sl” interface which is the interface used between two AllJoyn routers to exchange sessionless signals.
2. Register signal handlers to receive signals from “org.alljoyn.sl” interface.
3. Bind a well-known sessionless signal session port 100 to support incoming requests to fetch sessionless signals.

The consumer app interested in receiving sessionless signals registers a match rule with the AllJoyn router to receive sessionless signals. As a result, the SLS module initiates discovery for sessionless providers via the Name Service (either the legacy Name Service or NGNS based on the router version). On the provider side, the app sends a sessionless signal to the AllJoyn router. The SLS module stores the signal in a local message cache. The provider side sessionless signal generates a sessionless signal-specific well-known name and advertises that over AllJoyn network.

Upon discovering the sessionless signal provider, the consumer AllJoyn router establishes a session with the provider side AllJoyn router over the well-known session port for sessionless signals. Once the session is established, the consumer SLS module fetches sessionless signals via the org.alljoyn.sl interface.

The following sections detail the provider and consumer sessionless signal-related behavior.

7.1.1 Sessionless signal enhancement in the 14.06 release

Prior to the 14.06 release, the consumer side SLS module provides the functionality of requesting sessionless signals by matching certain filtering criteria specified by the AddMatch rule. The consumer fetches sessionless signals from all the providers and applies the AddMatch rules to filter received sessionless signals before passing these to interested applications.

The sessionless signal design in the 14.06 release was enhanced to enable a consumer application to request sessionless signals from provider applications supporting certain AllJoyn interfaces. For example, a Lighting Controller app can request Announcement sessionless signals only from those provider apps that implement the org.alljoyn.LightBulb interface.

The following key sessionless signal enhancements made to achieve this functionality:

- The sessionless signal advertised name was enhanced to add <INTERFACE> information from the header of the sessionless signal. Consumers use this to fetch sessionless signals only from those providers that are emitting signals from <INTERFACE> specified in the consumer side match rules. Multiple sessionless signal names are advertised, one for each unique interface in the sessionless signal cache.
- The match rule definition for AddMatch has been extended to add a new 'implements' key that can be used to indicate the desire to receive the Announcement sessionless signal only from applications implementing certain AllJoyn interfaces as specified in the application's Announcement signal.

Sessionless signals are only fetched from those providers that support interface details specified in the match rules. The AddMatch match rules are passed to the providers to filter signals based on those match rules.

7.2 Sessionless signal end-to-end logic

The sessionless signal end-to-end logic consists of the following aspects, which are detailed in the sections below.

- The provider caches signals and advertises the availability of signals.
- The consumer discovers sessionless signal providers.
- The consumer fetches sessionless signals from a provider.

7.2.1 Provider caches signals and advertises availability

On the provider side, the app sends a signal to the AllJoyn router with the SESSIONLESS flag set. The SLS module in the provider inserts the signal into its sessionless signal cache. The cache entry uses the combination of (SENDER, INTERFACE, MEMBER, and PATH) header fields of the signal as the key.

Subsequent sessionless signals sent to the AllJoyn router with the same (SENDER, INTERFACE, MEMBER, and PATH) header fields overwrite the already cached sessionless signal.

The provider AllJoyn router assigns a change_id for sessionless signals. The change_id is used to indicate updates to sessionless signals to consumers on the AllJoyn network. Each sessionless signal cache entry includes (SLS signal, change_id) tuple. The change_id is incremented by the provider only after a new signal is inserted into its cache and a consumer has requested signals from the provider since the last time the change_id was incremented.

The app may remove an entry from the provider's cache by calling the CancelSessionlessMessage method of the org.alljoyn.Bus interface of the /org/alljoyn/Bus object. The entry being removed is specified by the serial number of the signal. The change_id is not incremented when the provider removes a signal from its cache. The contents of the cache, including the associated change_ids, determine what the provider will advertise.

Prior to the AllJoyn 14.06 release, the SLS module requests and advertises the following sessionless signal well-known name:

- "org.alljoyn.sl.x<GUID>.x<change_id>"

  where:

  - GUID is the GUID of the AllJoyn router
  - change_id is the maximum change_id in the sessionless signal cache.

Starting with the AllJoyn 14.06 release, the SLS module requests and advertises the following sessionless signal well-known names:

- "org.alljoyn.sl.y<GUID>.x<change_id>"

  where:

  - GUID is the GUID of the AllJoyn router
  - change_id is the maximum change_id in the SLS cache.

- "<INTERFACE>.sl.y<GUID>.x<change_id>"

  where:

  - INTERFACE is the value of the INTERFACE header field of the signal
  - GUID is the GUID of the AllJoyn router
  - change_id is the maximum change_id for signals in the sessionless signal cache having the same value of the INTERFACE header field.

  At most, one well-known name is requested and advertised for each unique INTERFACE header field value in the sessionless signal cache.

Figure 7-2 shows the provider side SLS module logic prior to the AllJoyn 14.06 release.

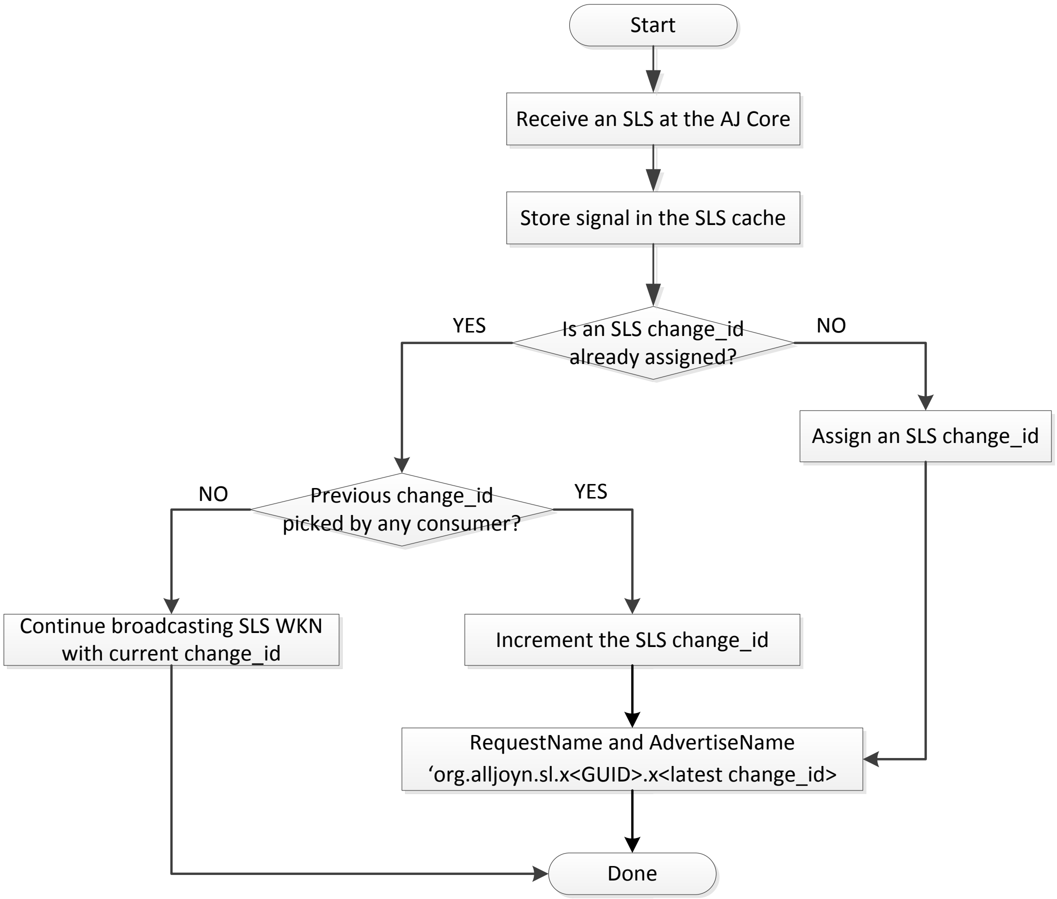

Start
Receive an SLS at the AJ Core
Store signal in the SLS cache
Is an SLS change_id already assigned?
YES
NO
Assign an SLS change_id
Previous change_id picked by any consumer?
NO
YES
Continue broadcasting SLS WKN with current change_id
Increment the SLS change_id
RequestName and AdvertiseName ‘org.alljoyn.sl.x<GUID>.x<latest change_id>
Done

Figure 7-2. Provider SLS module logic (prior to the AllJoyn 14.06 release)

Figure 7-3 shows the provider side SLS module logic introduced in the AllJoyn 14.06 release.

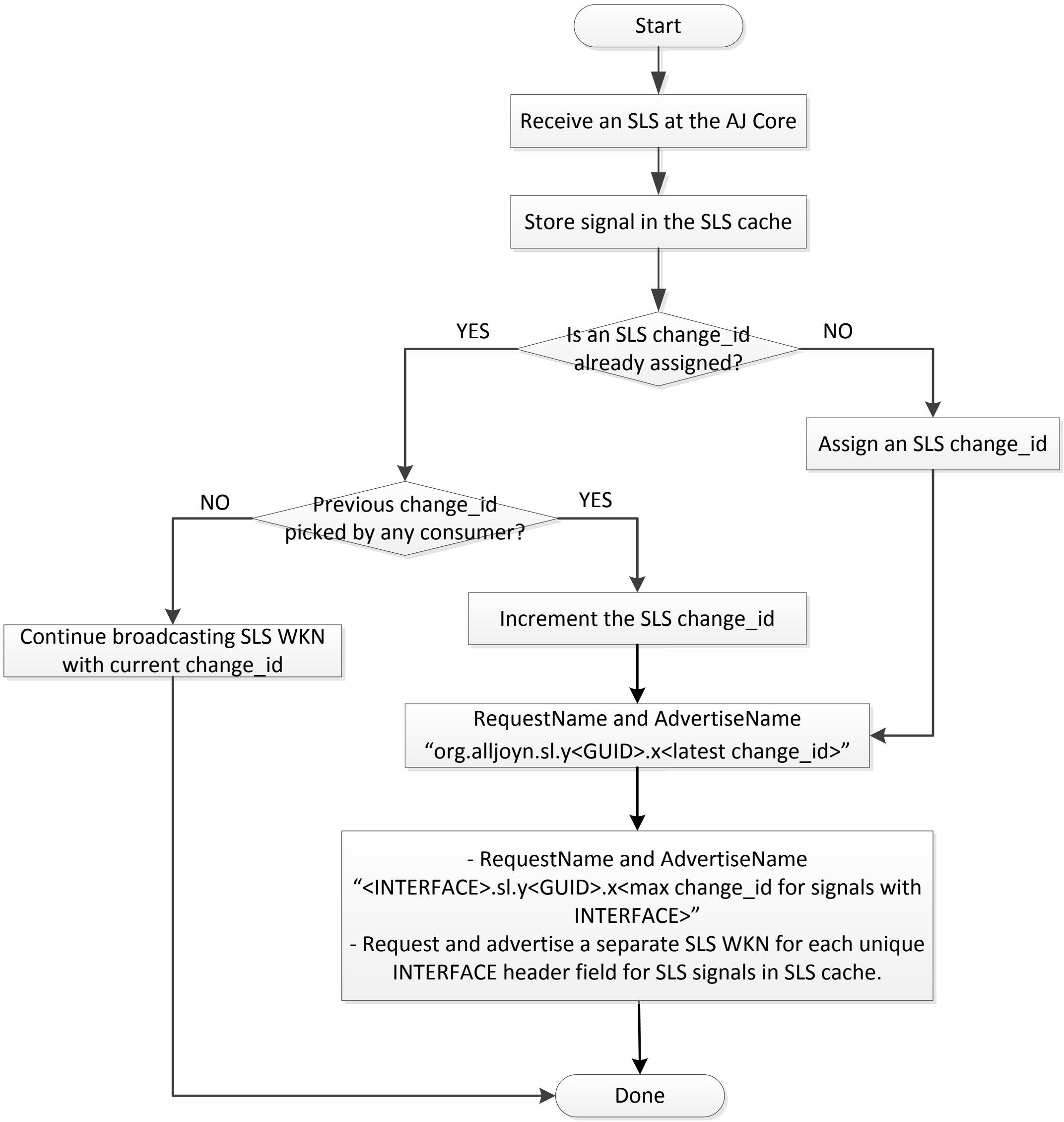

Start
Receive an SLS at the AJ Core
Store signal in the SLS cache
Is an SLS change_id already assigned?
YES
NO
Assign an SLS change_id
Previous change_id picked by any consumer?
NO
YES
Continue broadcasting SLS WKN with current change_id
Increment the SLS change_id
RequestName and AdvertiseName "org.alljoyn.sl.y<GUID>.x<latest change_id>"
- RequestName and AdvertiseName "<INTERFACE>.sl.y<GUID>.x<max change_id for signals with INTERFACE>"
- Request and advertise a separate SLS WKN for each unique INTERFACE header field for SLS signals in SLS cache.
Done

Figure 7-3. Provider SLS module logic (introduced in the AllJoyn 14.06 release)

7.2.2 Consumer discovers sessionless signal providers

On the consumer side, the app registers an interest in a sessionless signal by calling the D-Bus AddMatch method.[6] The match rule includes "sessionless='t'" to indicate registration for a sessionless signal, along with any other key/value pairs for filtering signals.

[6] The D-Bus AddMatch method is part of the org.freedesktop.The D-Bus interface and is implemented by the /org/freedesktop/DBus object.

Prior to the AllJoyn 14.06 release, the SLS module starts the name-based discovery process to discover the SLS WKN prefix “org.alljoyn.sl.” after receiving the first sessionless match rule. When the last match rule is removed by an app connected to the AllJoyn router, the SLS module cancels discovering the SLS WKN prefix.

Starting with the AllJoyn 14.06 release, the sessionless signal logic on the consumer side was enhanced to enable an application requesting sessionless signals from providers implementing certain AllJoyn interfaces. A new ‘implements’ key was added to the AddMatch method to achieve this. The ‘implements’ key specifies an AllJoyn interface that should be implemented by the sessionless signal provider.

Multiple ‘implements’ key/value pairs may be specified in a single match rule. These are treated as a logical AND when discovering sessionless signal providers. In the current implementation, the ‘implements’ key is only applicable for receiving the Announcement sessionless signal. An AddMatch that includes the ‘implements’ key should always include “interface= org.alljoyn.About”.

Figure 7-4 shows the consumer logic for discovering sessionless signal providers implemented in the 14.06 release.

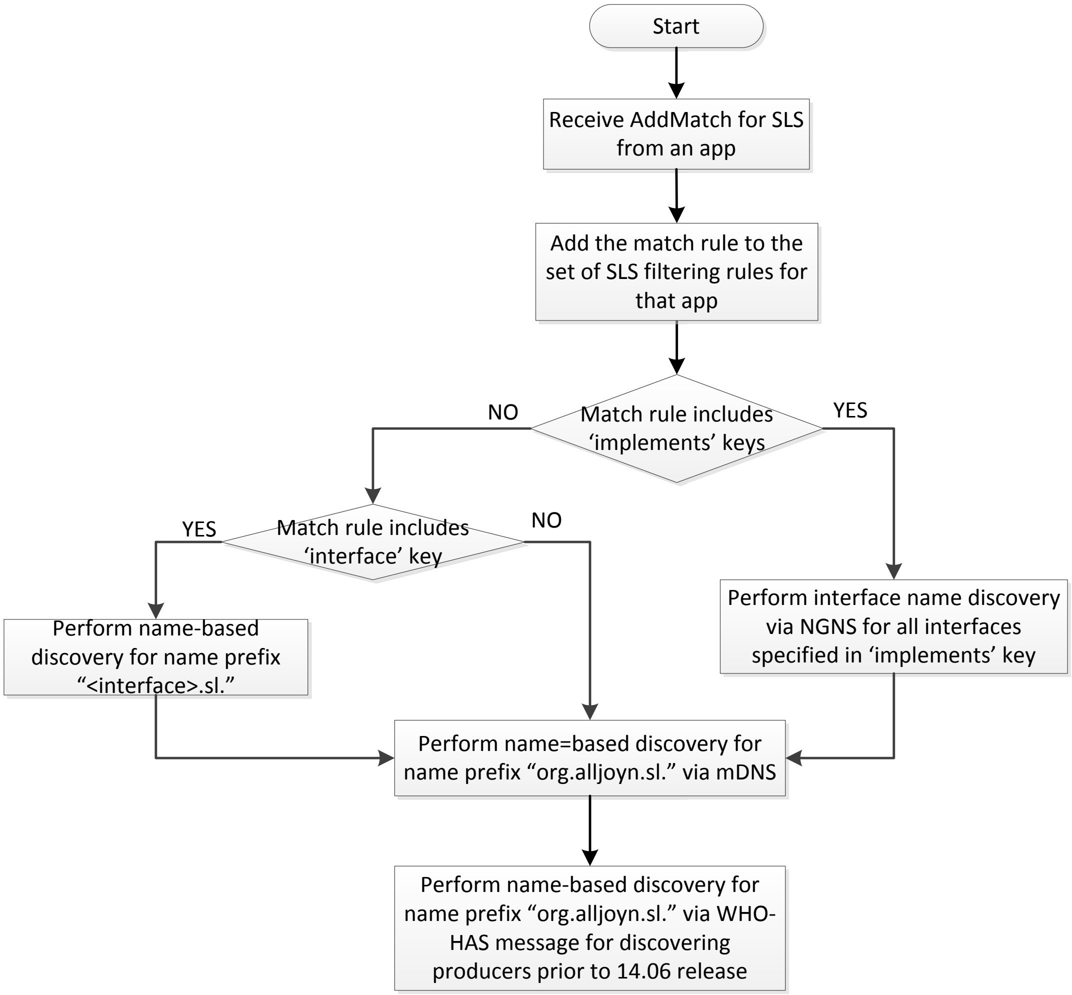

Start
Receive AddMatch for SLS from an app
Add the match rule to the set of SLS filtering rules for that app
Match rule includes 'implements' keys
NO
YES
Match rule includes 'interface' key
YES
NO
Perform name-based discovery for name prefix "<interface>.sl."
Perform interface name discovery via NGNS for all interfaces specified in 'implements' key
Perform name=based discovery for name prefix "org.alljoyn.sl." via mDNS
Perform name-based discovery for name prefix "org.alljoyn.sl." via WHO-HAS message for discovering producers prior to 14.06 release

Figure 7-4. Consumer logic for discovering sessionless providers (introduced in the AllJoyn 14.06 release)

If the AddMatch includes the 'interface' key but no 'implements' key, the SLS module performs name-based discovery to discover the name prefix "<interface>.sl." for the specified interface in the match rule.

If the AddMatch includes one or more 'implements' keys, the SLS module performs interface name discovery via the NGNS to discover providers implementing interfaces specified by the 'implements' key. The SLS module also performs a name-based discovery for name prefix "org.alljoyn.sl." both via mDNS and using the WHO-HAS message. The latter is intended to discover any sessionless signal providers prior to the 14.06 release. Name-based discovery over mDNS is performed to catch unsolicited mDNS responses from providers for new/updated sessionless signals.

The app unregisters interest in a sessionless signal by calling the RemoveMatch method with a previously added match rule. When the last sessionless signal match rule is removed, the SLS module stops sessionless signal-related discovery, including both name-based and interface name discovery.

7.2.3 Consumer fetches sessionless signals from a provider

After discovering a provider, the consumer SLS module determines if it needs to fetch sessionless signals from the provider based on the consumer's match rules and the change_id of the provider in the advertisement. The consumer SLS module keeps track of last acquired change_id for every provider GUID it has discovered via the sessionless signal advertised name. The sessionless signal fetch logic and the state maintained by the consumer for each provider was modified in the 14.06 release. The fetch logic functionality differences are described below.

- Prior to the 14.06 release, the consumer SLS module maintains the List<Provider GUID, last acquired change_id> information for discovered providers. The consumer SLS module fetch sessionless signals if an updated change_id is received from the provider as part of the sessionless signal advertised name.
- Starting with the 14.06 release, multiple sessionless signal advertised names can be received from a given provider. In addition, the associated change_id can be different for each of those sessionless signal advertised names. The consumer SLS module maintains the List<Provider GUID, SLS name, change_id> for discovered providers. It also keeps track of the match rules that have been applied for a given provider.

Figure 7-5 captures the consumer sessionless signal fetch logic implemented in the 14.06 release.

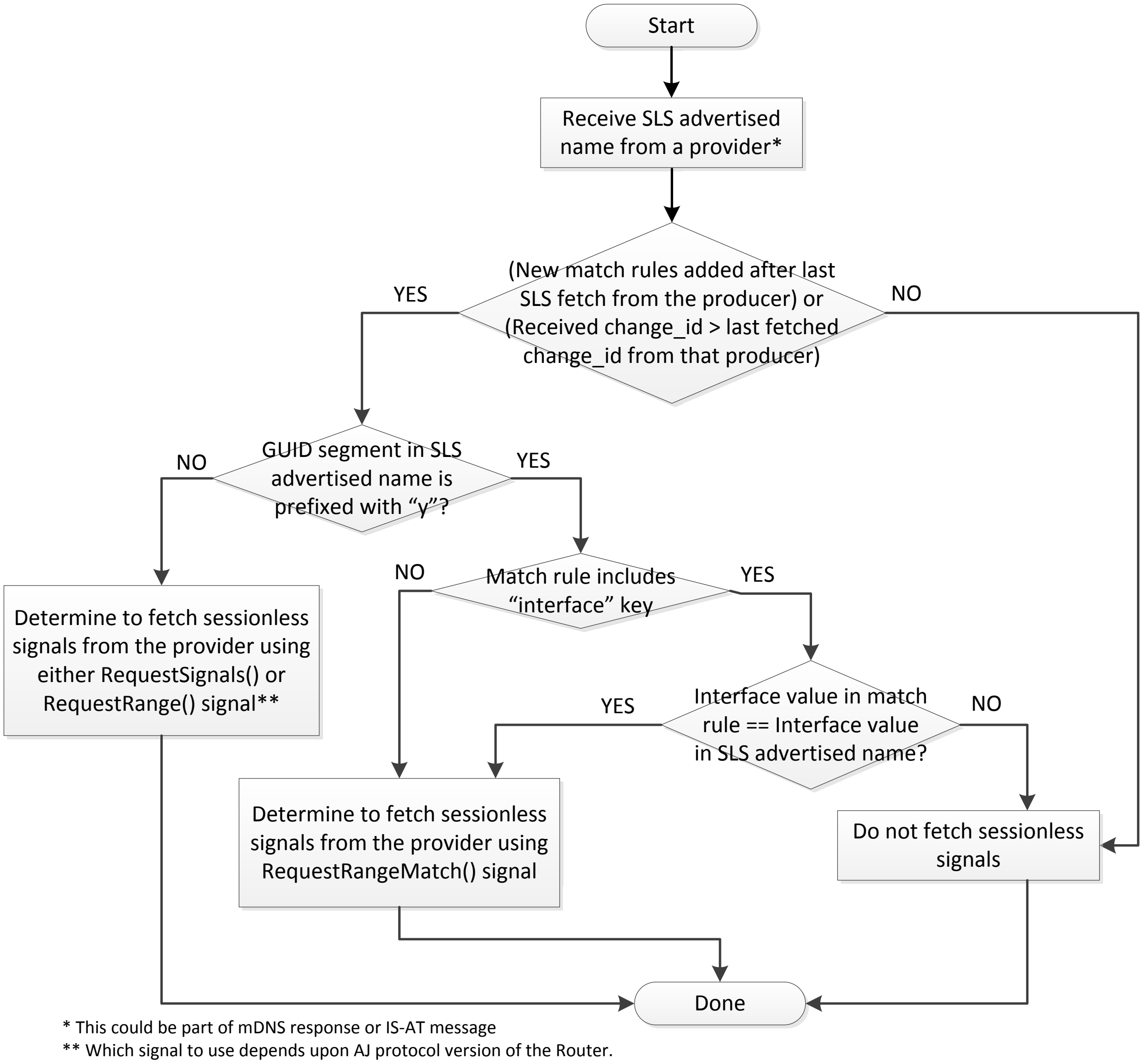

Start
Receive SLS advertised name from a provider*
(New match rules added after last SLS fetch from the producer) or (Received change_id > last fetched change_id from that producer)
YES
NO
GUID segment in SLS advertised name is prefixed with "y"?
NO
YES
Determine to fetch sessionless signals from the provider using either RequestSignals() or RequestRange() signal**
Match rule includes "interface" key
NO
YES
Interface value in match rule == Interface value in SLS advertised name?
YES
NO
Determine to fetch sessionless signals from the provider using RequestRangeMatch() signal
Do not fetch sessionless signals
Done
* This could be part of mDNS response or IS-AT message
** Which signal to use depends upon AJ protocol version of the Router.

Figure 7-5. Consumer logic to determine sessionless signal fetch (introduced in the 14.06 release)

The consumer SLS module receives the sessionless signal advertised name from a provider. The sessionless signal name can be received via multiple means:

- Solicited mDNS response over unicast as a result of an mDNS query
- Unsolicited mDNS response over multicast
- IS-AT multicast message (solicited or unsolicited).

The consumer checks if any new match rules have been added after the last fetch from the provider or the received change_id in the sessionless signal name is greater than what was fetched previously from that provider.

If either condition is true, the consumer initiates a sessionless signal fetch from a legacy provider (prior to the 14.06 release) without further checks. This is determined by examining the GUID segment of the sessionless signal advertised name received from the provider:

- Prior to the 14.06 release, the GUID is prefixed with an “x”.
- Starting with the 14.06 release, GUID is prefixed with a “y”.

If the sessionless signal name is from a 14.06 release or later provider, the consumer performs further checks to determine if the sessionless signal fetch should be done with the provider.

Starting with the 14.06 release, the sessionless signal name also includes the interface value from the sessionless signal header. In this case, the consumer SLS module checks whether the interface specified (if any) in the match rule is the same as the interface received in the sessionless signal name.

- If yes, the consumer initiates a sessionless signal fetch with the provider.
- If no interface was specified in the match rule, the consumer initiates a sessionless signal fetch anyway because this constitutes a wildcard match.

In the 14.06 release, a new RequestRangeMatch() signal is defined as part of the org.alljoyn.sl interface. This signal is used to fetch a set of sessionless signals that matches any of the match rules specified in the RequestRangeMatch() signal. The consumer SLS module uses this signal to fetch sessionless signals from the 14.06 providers.[7]

Whenever a new AddMatch rule is added for sessionless signals, the consumer SLS module triggers a catchup fetch with already known providers for the new match rule per the logic captured in Figure 7-5. For a catchup fetch, the RequestRangeMatch signal only includes the new match rule. If the new AddMatch includes an ‘implements’ key, the consumer SLS module performs discovery for providers implementing those interfaces.

The consumer schedules a sessionless signal fetch immediately for sessionless signal advertised names received in the solicited mDNS or IS-AT response messages. For sessionless signal advertised names received as part of the unsolicited mDNS or IS-AT response messages, the sessionless signal fetch is scheduled following a backoff algorithm as described in section 7.2.3.1.

The steps to fetch sessionless signals follow.

1. The consumer SLS module joins a session with the sessionless signal advertised name and known SLS port (port=100).
2. The consumer sends one of the signals defined in the org.alljoyn.sl (RequestSignals(), RequestRange() or RequestRangeMatch()) to the provider over established session to fetch sessionless signals.

[7] The current implementation does a catchup fetch for new match rules before fetching new signals for updated change_id. This results in two fetches upon receiving the sessionless signal advertised name, however, this would be a rare occurrence because adding a new match rule and receiving a sessionless signal advertised name typically does not occur at the same time.

3. The provider receives the request signal, sends the requested sessionless signals and leaves the session.

For details of which signals the provider sends from its cache to the consumer, see the definition of these signals in section 7.5.

The consumer receives the sessionless signals, and filters and routes them according to its match rules. The consumer SLS module maintains the information about the match rules applied and change_id fetched from the provider for future sessionless signal fetches.

7.2.3.1 Consumer sessionless signal fetch backoff algorithm

After determining that it needs to fetch sessionless signals from a given provider as per the logic described above, the consumer SLS module follows backoff logic to establish the session for the sessionless signal fetch from the provider. The consumer SLS module delays the join session attempt using the following algorithm:

- If this is the first join session attempt, delay randomly between [0, 1500] msec.
- For each subsequent attempt, delay randomly between 0 and half of the previous attempt, with a minimum interval of 250 milliseconds.

This produces the following delay intervals: 1500, 750, 375, 250, 250, … msec.

7.3 Sessionless signal message sequences (prior to the 14.06 release)

Since the sessionless logic has changed quite a bit in the 14.06 release, separate SLS message sequences are captured prior to the 14.06 release and starting from the 14.06 release.

The following use cases detail sessionless signal logic scenarios prior to the 14.06 release:

- First sessionless signal delivery
- Another AddMatch done by the same app
- Another app requests sessionless signals

7.3.1 First sessionless signal delivery

Figure 7-6 shows the message flow for the use case for sending and receiving the first sessionless signal on the provider and consumer respectively.

NOTE

The sessionless signal change_id is not carried in any of the sessionless signal messages. However, the provider AllJoyn router logic ensures that it only sends sessionless signal messages up to the change_id that was discovered by the consumer. When the JoinSession is done by the consumer, it uses the sessionless signal well-known name discovered from the IS-AT message. The change_id included in that well-known name provides the upper limit for the change_id for sending sessionless signals to the consumer.

A similar message flow is applicable for the use case when a subsequent sessionless signal is delivered. The main difference is that the provider updates the change_id, if applicable, as per the change_id update logic described above.

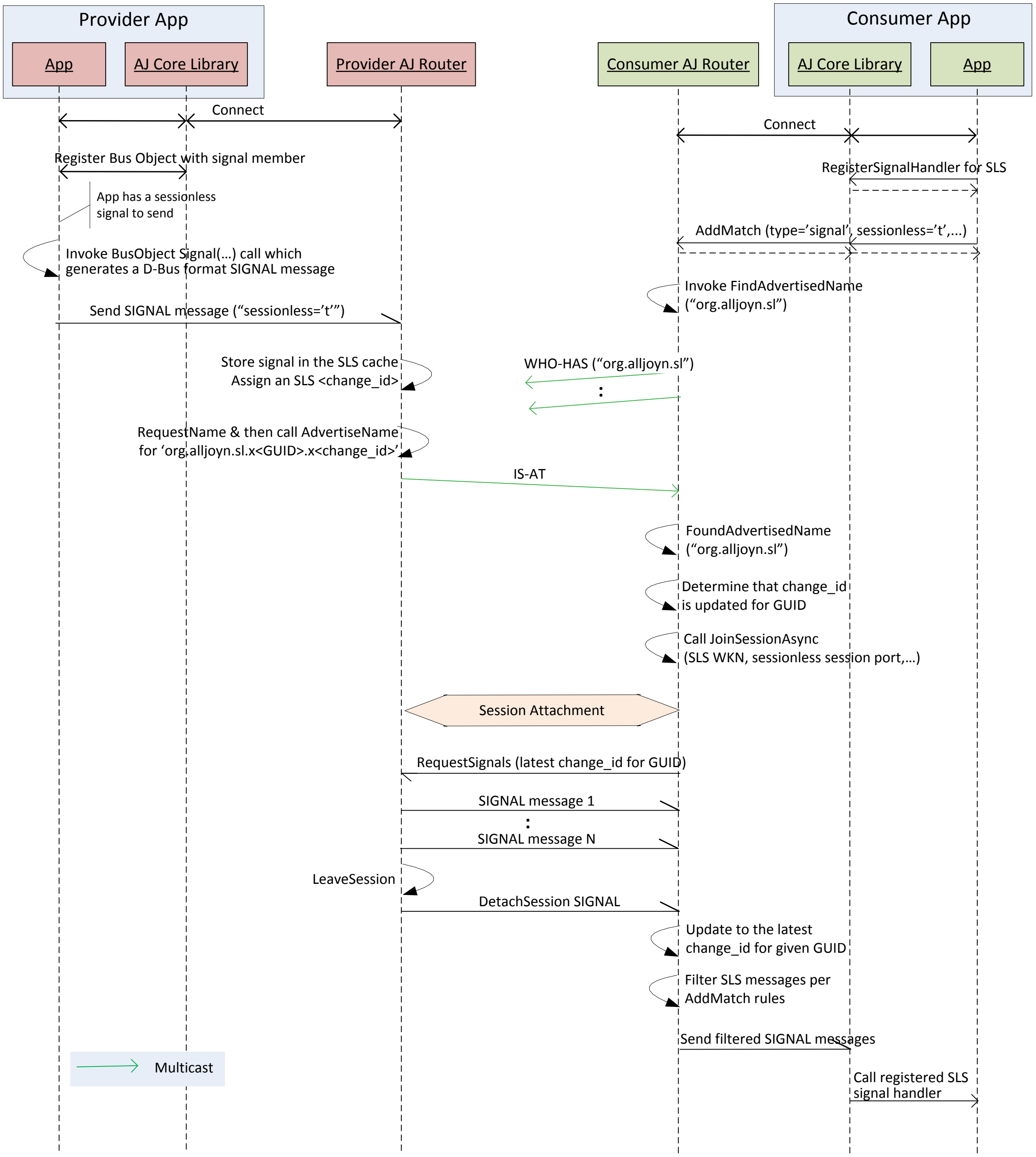

Provider App
App
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer App
AJ Core Library
App
Connect
Connect
Register Bus Object with signal member
RegisterSignalHandler for SLS
App has a sessionless signal to send
AddMatch (type='signal', sessionless='t',...)
Invoke BusObject Signal(...) call which generates a D-Bus format SIGNAL message
Invoke FindAdvertisedName ("org.alljoyn.sl")
Send SIGNAL message ("sessionless='t'")
Store signal in the SLS cache
Assign an SLS <change_id>
WHO-HAS ("org.alljoyn.sl")
:
RequestName & then call AdvertiseName for 'org.alljoyn.sl.x<GUID>.x<change_id>'
IS-AT
FoundAdvertisedName ("org.alljoyn.sl")
Determine that change_id is updated for GUID
Call JoinSessionAsync (SLS WKN, sessionless session port,...)
Session Attachment
RequestSignals (latest change_id for GUID)
SIGNAL message 1
:
SIGNAL message N
LeaveSession
DetachSession SIGNAL
Update to the latest change_id for given GUID
Filter SLS messages per AddMatch rules
Send filtered SIGNAL messages
Multicast
Call registered SLS signal handler

Figure 7-6. First sessionless signal delivery

The message flow steps are described below.

1. Both provider and consumer apps connect to the AllJoyn router, and a unique name is assigned to the app endpoint by the AllJoyn router.

2. The provider app registers its service object implementing an interface containing signal member with the AllJoyn core library.
3. The consumer app registers a signal handler for the sessionless signal by invoking the AllJoyn core library's RegisterSignalHandler API.
4. The consumer app invokes AllJoyn core library's AddMatch API to add a rule for receiving sessionless signals. The API specifies a signal match rule with type='signal', sessionless='t' and other applicable parameters.
5. The AllJoyn core library invokes the AllJoyn router's AddMatch method to add the sessionless signal filtering rule at the AllJoyn router.
6. The consumer AllJoyn router invokes FindAdvertisedName() with the sessionless SLS WKN prefix "org.alljoyn.sl" to discover providers offering sessionless signals.
7. The consumer AllJoyn router sends a WHO-HAS message for "org.alljoyn.sl" prefix.
8. The provider app has a sessionless signal to send. It invokes the BusObject Signal(…) call which generates an AllJoyn SIGNAL message with the sessionless flag set to true.
9. The SIGNAL message is sent from the app to the AllJoyn router.
10. The provider AllJoyn router stores the signal in the sessionless signal cache and assigns a new sessionless signal change_id number.
11. The provider AllJoyn router generates a well-known name for the sessionless signal with the latest change_id using the format org.alljoyn.sl.x<GUID>.x<change_id>.
12. The provider AllJoyn router does a RequestName for this well-known name to reserve this name. It then calls the AdvertiseName method call to advertise this name on the AllJoyn network.
13. The provider AllJoyn router sends out an IS-AT message with the generated sessionless signal well-known name.
14. The consumer AllJoyn router receives the IS-AT message that passes the prefix matching for "org.alljoyn.sl". A FoundAdvertisedName signal gets generated for the the sessionless signal prefix.
15. The consumer AllJoyn router compares its current change_id for the AllJoyn router GUID in the IS-AT message with the change_id received in the IS-AT message. It determines that the received change_id is different than the current change_id and it needs to get the new set of sessionless signals from the provider AllJoyn router.
16. The consumer AllJoyn router invokes the JoinSessionAsync method to start a session with the provider AllJoyn router. It specifies the sessionless signal wel-known name and the sessionless session port among other parameters.

    This initiates a session attachment flow at the AllJoyn router level between the consumer and provider.
17. Once the session is established, the consumer AllJoyn router sends a RequestSignals signal to request the latest set of sessionless signals from the provider app. This signal includes the last acquired change_id for the GUID of provider AllJoyn router.

18. The provider AllJoyn router sends SIGNAL messages for all sessionless signals added after the change_id provided in the RequestSignals message. The SIGNAL messages have destination as local endpoint of the consumer AllJoyn router.
19. Once all SIGNAL messages have been sent, the provider AllJoyn router initiates a LeaveSession method for the connected session. This triggers sending a DetachSession SIGNAL message to the consumer AllJoyn router.
20. After receiving the DetachSession signal, the consumer AllJoyn router knows that it has received all new sessionless signals from the provider AllJoyn router. It then updates its change_id for that GUID to the latest received change_id from the IS_AT message.
21. The consumer AllJoyn router filters the received sessionless signal messages based on the registered AddMatch rules for the sessionless signals.
22. The consumer AllJoyn router sends SIGNAL messages to the AllJoyn core library via callback. The AllJoyn core library in turn calls the registered signal handler for the sessionless signal.

7.3.2 Another AddMatch done by the app

A consumer app might invoke a subsequent AddMatch call for sessionless signals. Prior to the 14.06 release, this is interpreted as adding another rule for filtering future received sessionless signals.

- At the AllJoyn router, any future received SLS messages will be filtered based on the combined set of match rules.
- If there is a more restrictive rule which matches a small set of sessionless signals and there is another less restrictive rule which matches a larger superset of signals, AllJoyn router will always send larger superset of signals to the app.

NOTE

The AllJoyn router does not re-fetch sessionless signal messages associated with the existing change_id when a new match rule is added. The new match rule is applied to any future received messages. Starting with the 14.06 release, this behavior was modified and the SLS module initiates the sessionless signal fetch whenever any new match rule is added.

Figure 7-7 shows the message flow for the scenario where another AddMatch is done by the same app. Most of the steps are same as the first sessionless signal delivery. The main difference is that no FindAdvertisedName needs to be done for “org.alljoyn.sl” because discovery for the SLS WKN prefix was already initiated at the time of receiving first AddMatch for the sessionless signal and is already in progress.

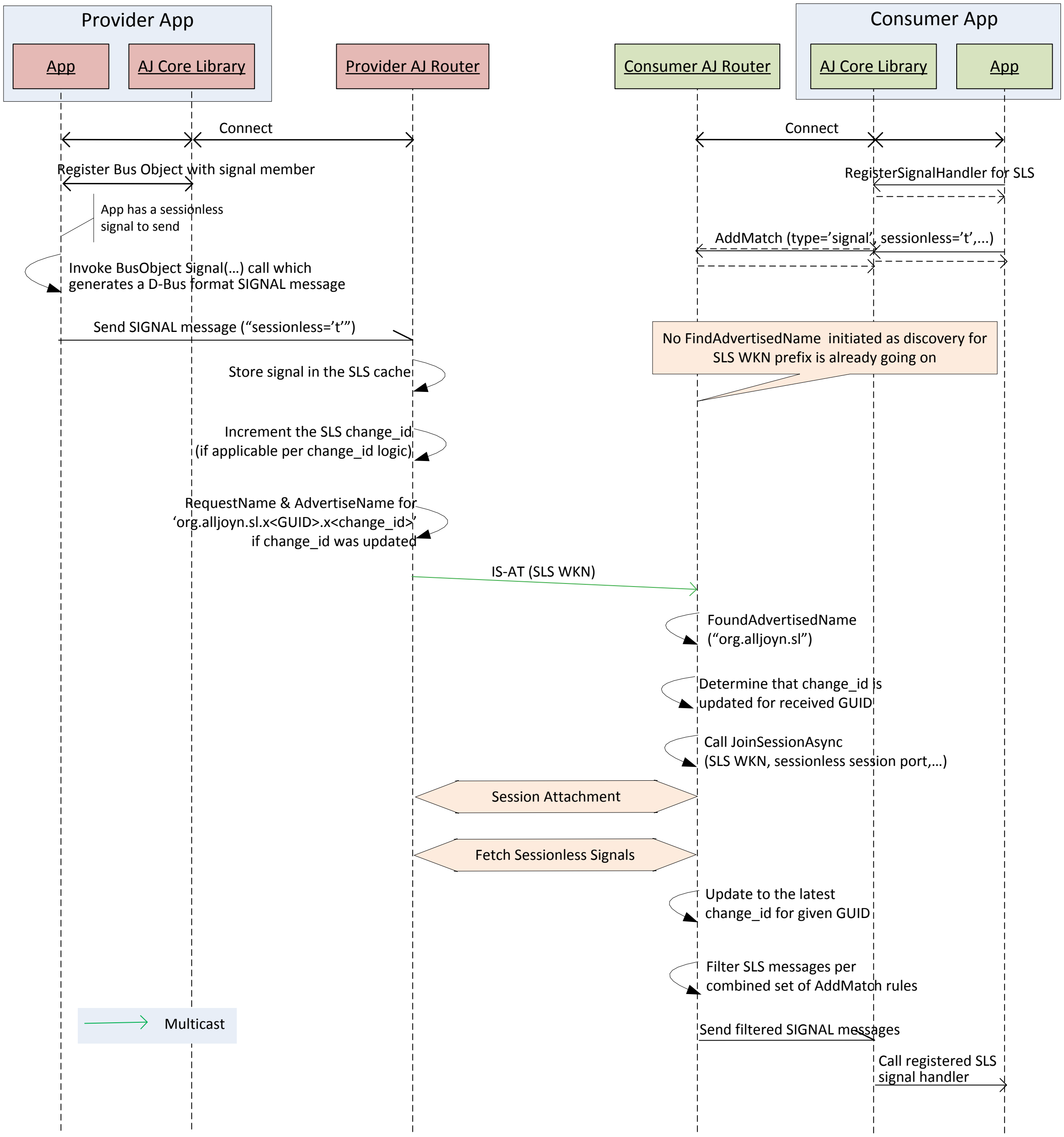

Provider App
App
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer App
AJ Core Library
App
Connect
Connect
Register Bus Object with signal member
RegisterSignalHandler for SLS
App has a sessionless signal to send
AddMatch (type='signal', sessionless='t',...)
Invoke BusObject Signal(...) call which generates a D-Bus format SIGNAL message
Send SIGNAL message ("sessionless='t'")
No FindAdvertisedName initiated as discovery for SLS WKN prefix is already going on
Store signal in the SLS cache
Increment the SLS change_id (if applicable per change_id logic)
RequestName & AdvertiseName for 'org.alljoyn.sl.x<GUID>.x<change_id>' if change_id was updated
IS-AT (SLS WKN)
FoundAdvertisedName ("org.alljoyn.sl")
Determine that change_id is updated for received GUID
Call JoinSessionAsync (SLS WKN, sessionless session port,...)
Session Attachment
Fetch Sessionless Signals
Update to the latest change_id for given GUID
Filter SLS messages per combined set of AddMatch rules
Multicast
Send filtered SIGNAL messages
Call registered SLS signal handler

Figure 7-7. Another AddMatch done by the app

7.3.3 Another app requesting for sessionless signal

For the use case where multiple apps are connected to a given AllJoyn router, each application can invoke the AddMatch call to add match rules for sessionless signals with the AllJoyn router. These AddMatch calls can be received at different times. When the first AddMatch call is received from an app, the AllJoyn router re-fetches the currently valid sessionless signals from the already discovered providers for sending to that app.

The AllJoyn router uses the RequestRange signal to fetch the current set of sessionless signals.

Figure 7-8 shows the message flow for the scenario when a subsequent app performs the first AddMatch for sessionless signals.

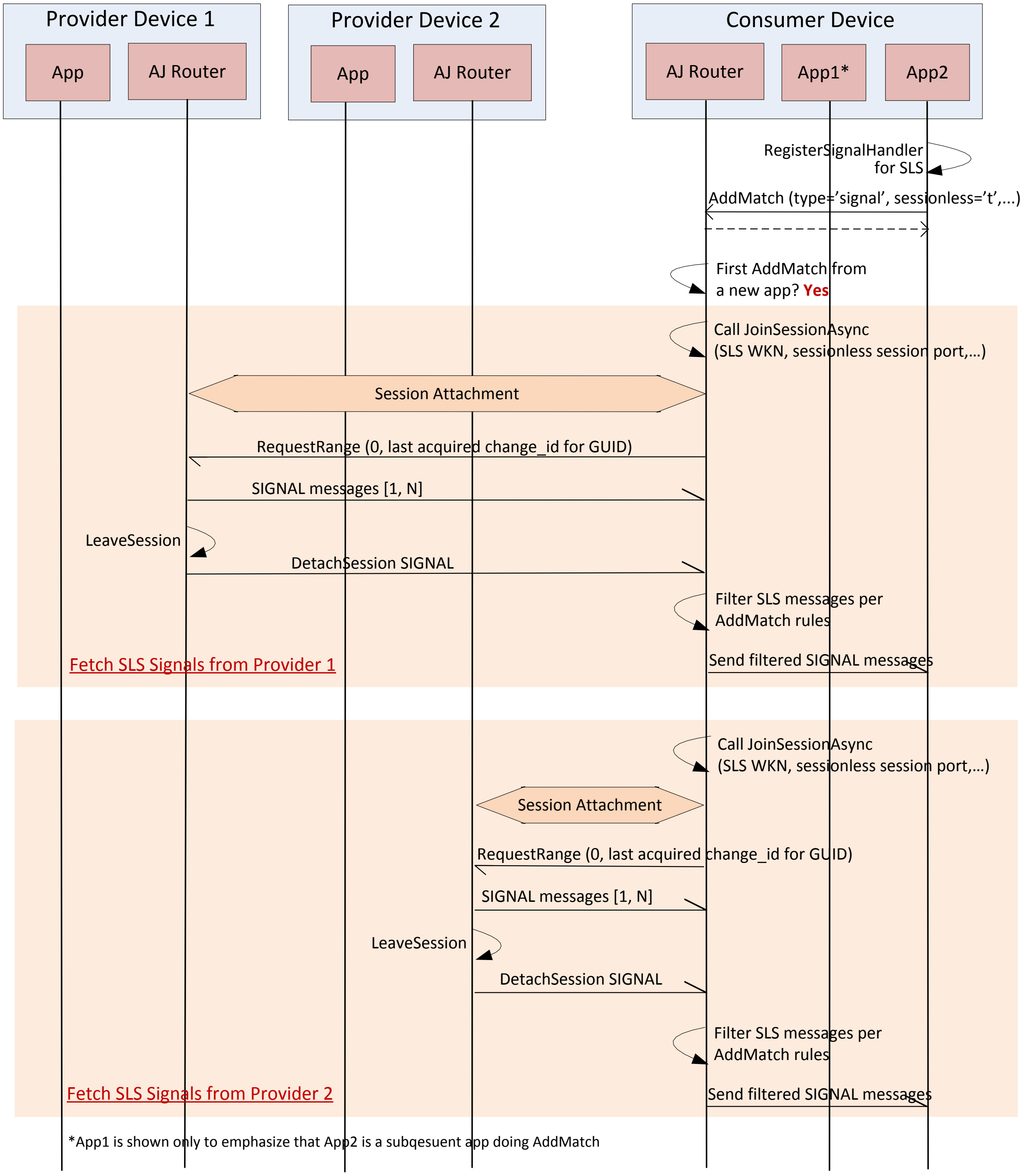

Provider Device 1
App
AJ Router
Provider Device 2
App
AJ Router
Consumer Device
AJ Router
App1*
App2
RegisterSignalHandler for SLS
AddMatch (type='signal', sessionless='t',...)
First AddMatch from a new app? Yes
Call JoinSessionAsync (SLS WKN, sessionless session port,...)
Session Attachment
RequestRange (0, last acquired change_id for GUID)
SIGNAL messages [1, N]
LeaveSession
DetachSession SIGNAL
Filter SLS messages per AddMatch rules
Send filtered SIGNAL messages
Fetch SLS Signals from Provider 1
Call JoinSessionAsync (SLS WKN, sessionless session port,...)
Session Attachment
RequestRange (0, last acquired change_id for GUID)
SIGNAL messages [1, N]
LeaveSession
DetachSession SIGNAL
Filter SLS messages per AddMatch rules
Send filtered SIGNAL messages
Fetch SLS Signals from Provider 2
*App1 is shown only to emphasize that App2 is a subqesuent app doing AddMatch

Figure 7-8. Another app requesting for sessionless signals

7.4 Sessionless signal message sequences (14.06 release)

In the 14.06 release, the sessionless signal logic was enhanced as described earlier in this chapter. This section captures the sessionless signal message sequences with the enhanced logic. The following use cases are detailed:

- First sessionless signal delivery
- Sessionless signal delivery between a new consumer and a legacy provider
- Sessionless signal delivery between a legacy consumer and a new provider
- Subsequent AddMatch done by an app

7.4.1 First sessionless signal delivery

This use case defines when the first AddMatch is done by a consumer app to receive sessionless signals. The match rules specified in the AddMatch may or may not include the new ‘implements’ key. Both use cases are captured accordingly:

- AddMatch does not include ‘implements’ key
- AddMatch includes ‘implements’ key

7.4.1.1 AddMatch does not include ‘implements’ key

Figure 7-9 shows the message flow for sending and receiving the first sessionless signal for the use case when AddMatch does not include an ‘implements’ key/value pair.

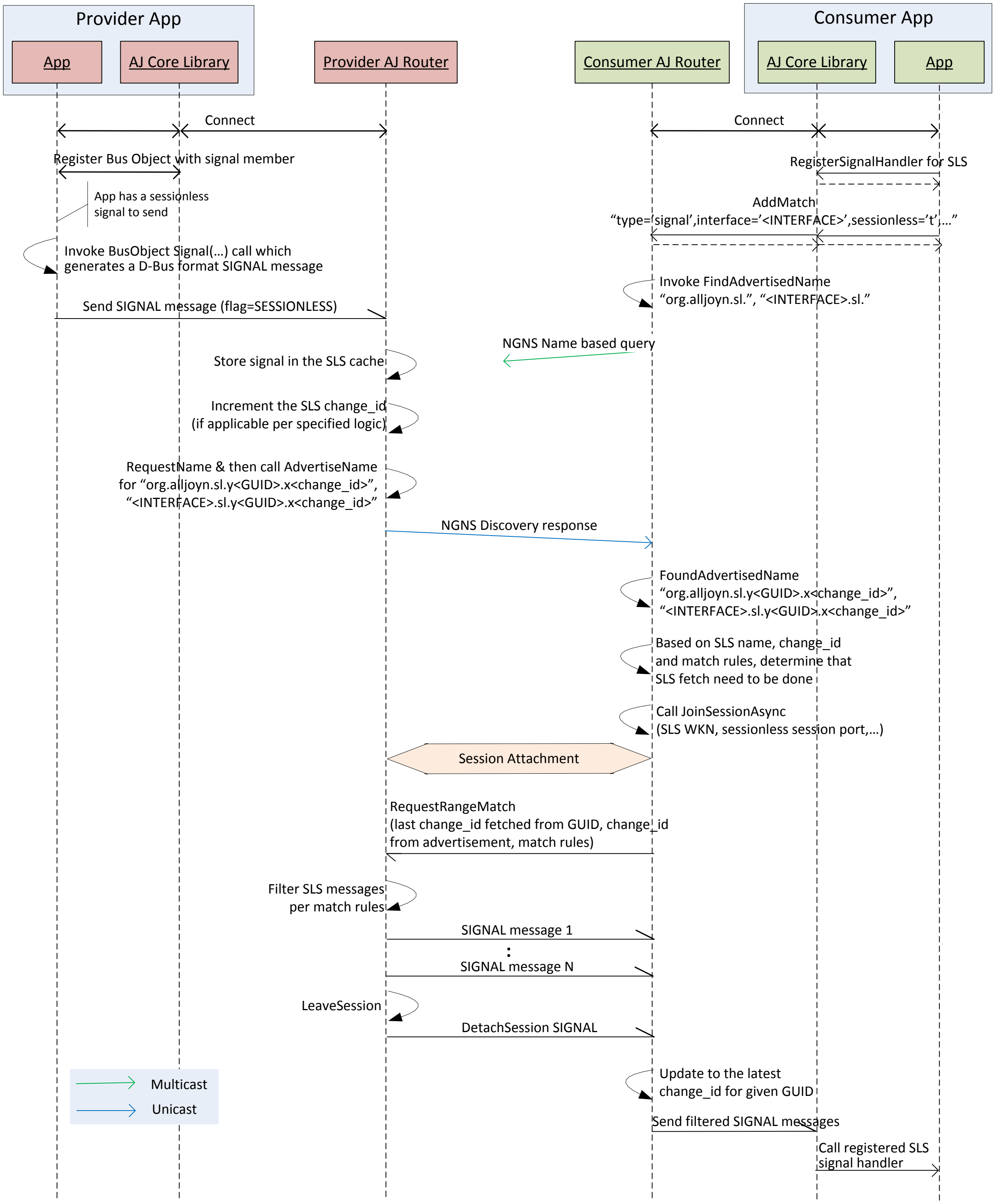

Provider App
App
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer App
AJ Core Library
App
Connect
Connect
Register Bus Object with signal member
RegisterSignalHandler for SLS
App has a sessionless signal to send
AddMatch
“type='signal',interface='<INTERFACE>',sessionless='t',...”
Invoke BusObject Signal(...) call which generates a D-Bus format SIGNAL message
Invoke FindAdvertisedName “org.alljoyn.sl.”, “<INTERFACE>.sl.”
Send SIGNAL message (flag=SESSIONLESS)
NGNS Name based query
Store signal in the SLS cache
Increment the SLS change_id (if applicable per specified logic)
RequestName & then call AdvertiseName for “org.alljoyn.sl.y<GUID>.x<change_id>”, “<INTERFACE>.sl.y<GUID>.x<change_id>”
NGNS Discovery response
FoundAdvertisedName “org.alljoyn.sl.y<GUID>.x<change_id>”, “<INTERFACE>.sl.y<GUID>.x<change_id>”
Based on SLS name, change_id and match rules, determine that SLS fetch need to be done
Call JoinSessionAsync (SLS WKN, sessionless session port,...)
Session Attachment
RequestRangeMatch (last change_id fetched from GUID, change_id from advertisement, match rules)
Filter SLS messages per match rules
SIGNAL message 1
:
SIGNAL message N
LeaveSession
DetachSession SIGNAL
Multicast
Unicast
Update to the latest change_id for given GUID
Send filtered SIGNAL messages
Call registered SLS signal handler

Figure 7-9. First sessionless signal delivery (no ‘implements’ key in AddMatch)

The message flow steps are described below.

1. Both the provider and consumer apps connect to the AllJoyn router.

2. The provider app registers its service object implementing an interface containing signal member with the AllJoyn core library.
3. The consumer app registers a signal handler for the sessionless signal by invoking the AllJoyn core library's RegisterSignalHandler API.
4. The consumer app invokes the AllJoyn core library's AddMatch API to add a rule for receiving sessionless signals. The API specifies a signal match rule with type='signal', interface='<INTERFACE>', sessionless='t' and other applicable parameters.
5. The AllJoyn core library invokes the AllJoyn router's AddMatch method to add the sessionless signal filtering rule at the AllJoyn router.
6. The consumer AllJoyn router invokes the FindAdvertisedName method with the SLS WKN prefix "org.alljoyn.sl." to discover providers prior to the 14.06 release and SLS WKN "<INTERFACE>sl." to discover new providers starting in the 14.06 release.
7. The consumer AllJoyn router sends out name-based query messages using the NGNS.
8. The provider app has a sessionless signal to send. It invokes the BusObject Signal(…) call which generates an AllJoyn SIGNAL message with SESSIONLESS flag set to true.
9. The SIGNAL message is sent from app to the AllJoyn router.
10. The provider AllJoyn router stores the signal in the sessionless signal cache.
11. The provider AllJoyn router generates the following:
    - A well-known name for the sessionless signal of the format "org.alljoyn.sl.y<GUID>.x<change_id>" with the latest change_id.
    - A second well-known name for the sessionless signal of the format "<INTERFACE>.y<GUID>.x<change_id>" with the latest change_id associated with signals generated by the given <INTERFACE>.
12. The provider AllJoyn router invokes the RequestName and AdvertiseName method calls to request and advertise these names.
13. The provider AllJoyn router sends out a Name Service response messages to advertise these sessionless signal well-known names using NGNS.
14. The consumer AllJoyn router receives the Name Service response messages which pass the prefix matching for "org.alljoyn.sl." or "<INTERFACE>.y<GUID>.x<change_id>". A FoundAdvertisedName signal gets generated for the advertisements.
15. The consumer determines that it needs to fetch sessionless signals from the provider, based on the received sessionless signal advertised name, change_id comparison, and current set of match rules as per logic described in section 7.2.3.
16. The consumer AllJoyn router invokes JoinSessionAsync method call to start a session with the provider AllJoyn router. It specifies the sessionless signal well-known name and sessionless session port among other parameters.

This initiates a session attachment flow at the AllJoyn router level between the consumer and provider.

17. Once the session is established, the consumer AllJoyn router sends the RequestRangeMatch signal to request the latest set of sessionless signals from the provider device. This signal includes last acquired change_id for the GUID of provider AllJoyn router, and the match rules that have not yet been applied to the provider GUID.
18. The provider AllJoyn router sends the AllJoyn SIGNAL messages for all sessionless signals added that pass the match rules in the range provided in the RequestRangeMatch signal.
19. Once all SIGNAL messages have been sent, the provider AllJoyn router initiates LeaveSession for the connected session. This triggers sending a DetachSession SIGNAL message to the consumer AllJoyn router.
20. After receiving the DetachSession signal, the consumer AllJoyn router knows that it has received all new sessionless signals from the provider AllJoyn router. It then updates its change_id for that GUID to the latest received change_id from the advertisement and also updates status for match rules which have been applied to that provider.
21. The consumer AllJoyn router sends SIGNAL messages to the AllJoyn core library via callback. The AllJoyn core library in turn calls the registered signal handler for the sessionless signal.

7.4.1.2 AddMatch includes the ‘implements’ key

Figure 7-10 shows the message flow for sending and receiving of the first sessionless signal for the use case when AddMatch includes one or more ‘implements’ key/value pair.

NOTE

Currently, the ‘implements’ key only applies to the Announcement sessionless signal. As a result, the AddMatch in this case must include “interface=org.alljoyn.About”, which is the interface that emits the Announcement signal.

The AllJoyn router initiates interface name discovery via NGNS to find providers which implements the specified interfaces. Once those providers are discovered, the consumer AllJoyn router fetches the Announcement signals from those providers.

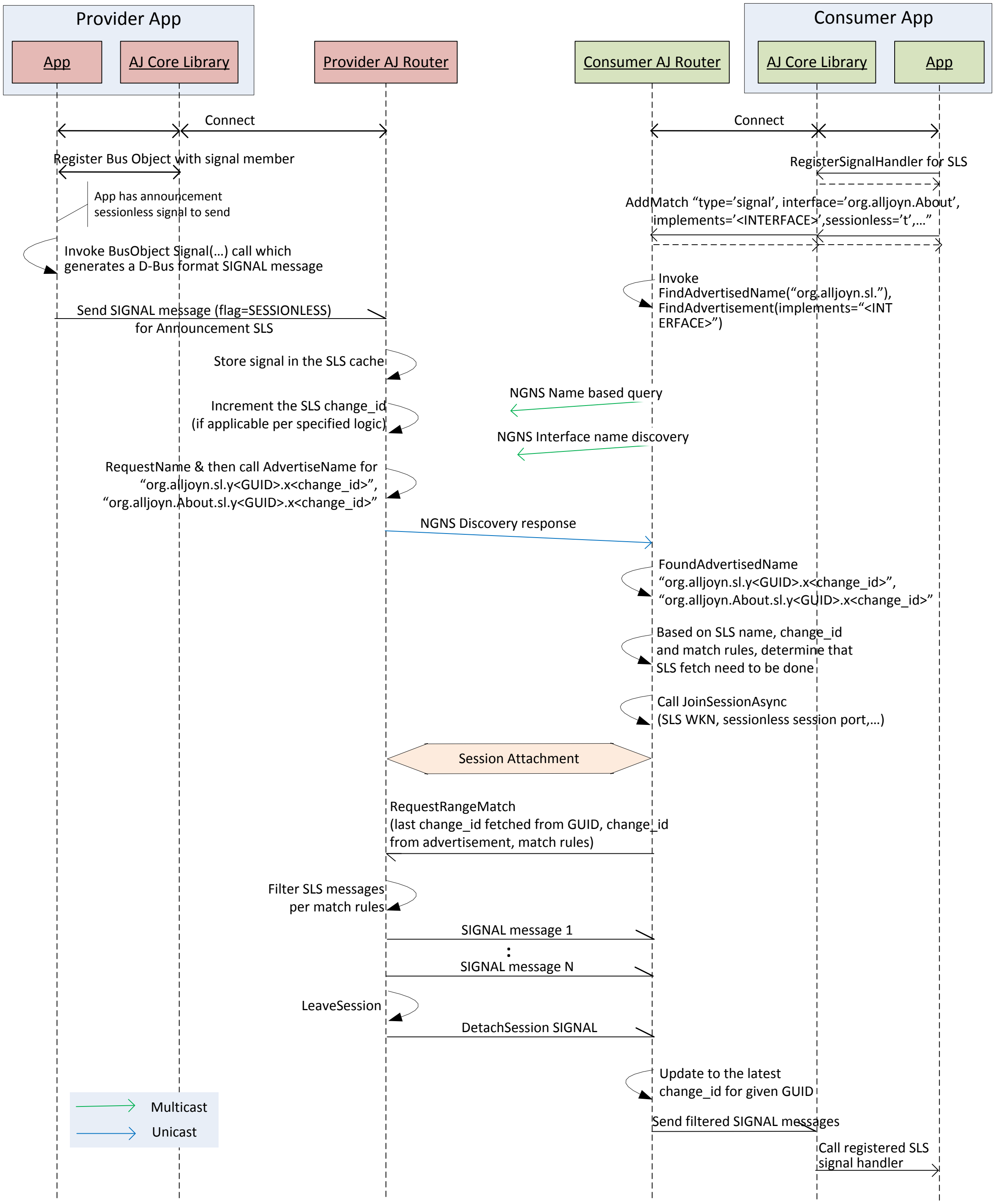

Provider App
App
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer App
AJ Core Library
App
Connect
Connect
Register Bus Object with signal member
RegisterSignalHandler for SLS
App has announcement
sessionless signal to send
AddMatch “type=’signal’, interface=’org.alljoyn.About’,
implements=’<INTERFACE>’,sessionless=’t’,...”
Invoke BusObject Signal(…) call which
generates a D-Bus format SIGNAL message
Invoke
FindAdvertisedName(“org.alljoyn.sl.”),
FindAdvertisement(implements=“<INT
ERFACE>”)
Send SIGNAL message (flag=SESSIONLESS)
for Announcement SLS
Store signal in the SLS cache
NGNS Name based query
Increment the SLS change_id
(if applicable per specified logic)
NGNS Interface name discovery
RequestName & then call AdvertiseName for
“org.alljoyn.sl.y<GUID>.x<change_id>”,
“org.alljoyn.About.sl.y<GUID>.x<change_id>”
NGNS Discovery response
FoundAdvertisedName
“org.alljoyn.sl.y<GUID>.x<change_id>”,
“org.alljoyn.About.sl.y<GUID>.x<change_id>”
Based on SLS name, change_id
and match rules, determine that
SLS fetch need to be done
Call JoinSessionAsync
(SLS WKN, sessionless session port,…)
Session Attachment
RequestRangeMatch
(last change_id fetched from GUID, change_id
from advertisement, match rules)
Filter SLS messages
per match rules
SIGNAL message 1
:
SIGNAL message N
LeaveSession
DetachSession SIGNAL
Update to the latest
change_id for given GUID
Send filtered SIGNAL messages
Call registered SLS
signal handler
Multicast
Unicast

Figure 7-10. First sessionless signal delivery (with ‘implements’ key in AddMatch)

7.4.2 Sessionless signal delivery between a new consumer and a legacy provider

Figure 7-11 shows the message flow when a new consumer (14.06 release or later) received sessionless signals from a legacy provider (prior to the 14.06 release).

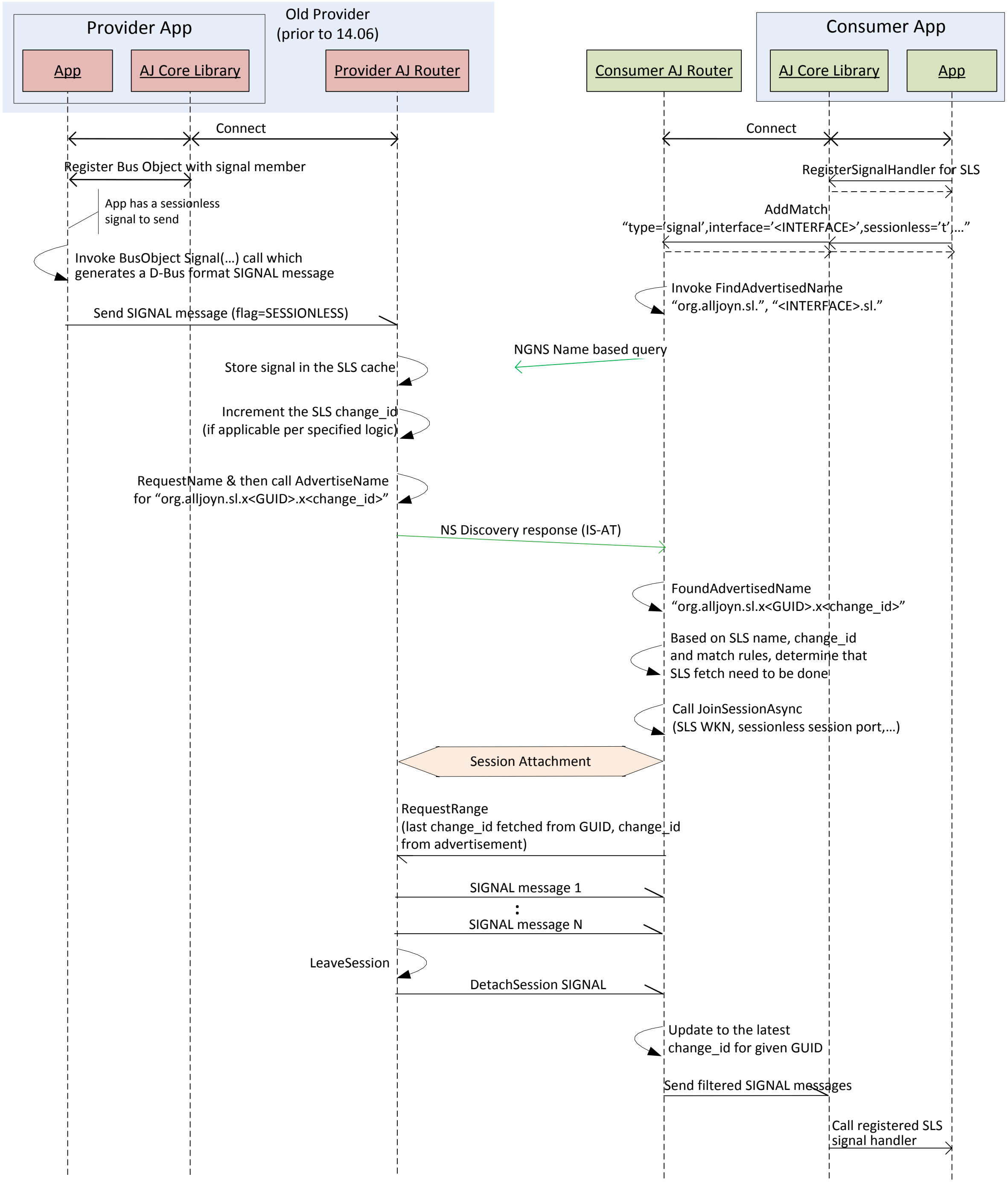

Old Provider
(prior to 14.06)
Provider App
App
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer App
AJ Core Library
App
Connect
Connect
Register Bus Object with signal member
RegisterSignalHandler for SLS
App has a sessionless
signal to send
AddMatch
“type='signal',interface='<INTERFACE>',sessionless='t',...”
Invoke BusObject Signal(...) call which
generates a D-Bus format SIGNAL message
Invoke FindAdvertisedName
“org.alljoyn.sl.”, “<INTERFACE>.sl.”
Send SIGNAL message (flag=SESSIONLESS)
NGNS Name based query
Store signal in the SLS cache
Increment the SLS change_id
(if applicable per specified logic)
RequestName & then call AdvertiseName
for “org.alljoyn.sl.x<GUID>.x<change_id>”
NS Discovery response (IS-AT)
FoundAdvertisedName
“org.alljoyn.sl.x<GUID>.x<change_id>”
Based on SLS name, change_id
and match rules, determine that
SLS fetch need to be done
Call JoinSessionAsync
(SLS WKN, sessionless session port,...)
Session Attachment
RequestRange
(last change_id fetched from GUID, change_id
from advertisement)
SIGNAL message 1
:
SIGNAL message N
LeaveSession
DetachSession SIGNAL
Update to the latest
change_id for given GUID
Send filtered SIGNAL messages
Call registered SLS
signal handler

Figure 7-11. Sessionless signal logic between new consumer and legacy provider

7.4.3 Sessionless signal delivery between a legacy consumer and a new provider

Figure 7-12 shows the message flow when a legacy consumer (prior to the 14.06 release) is receiving sessionless signals from a new provider (14.06 release or later).

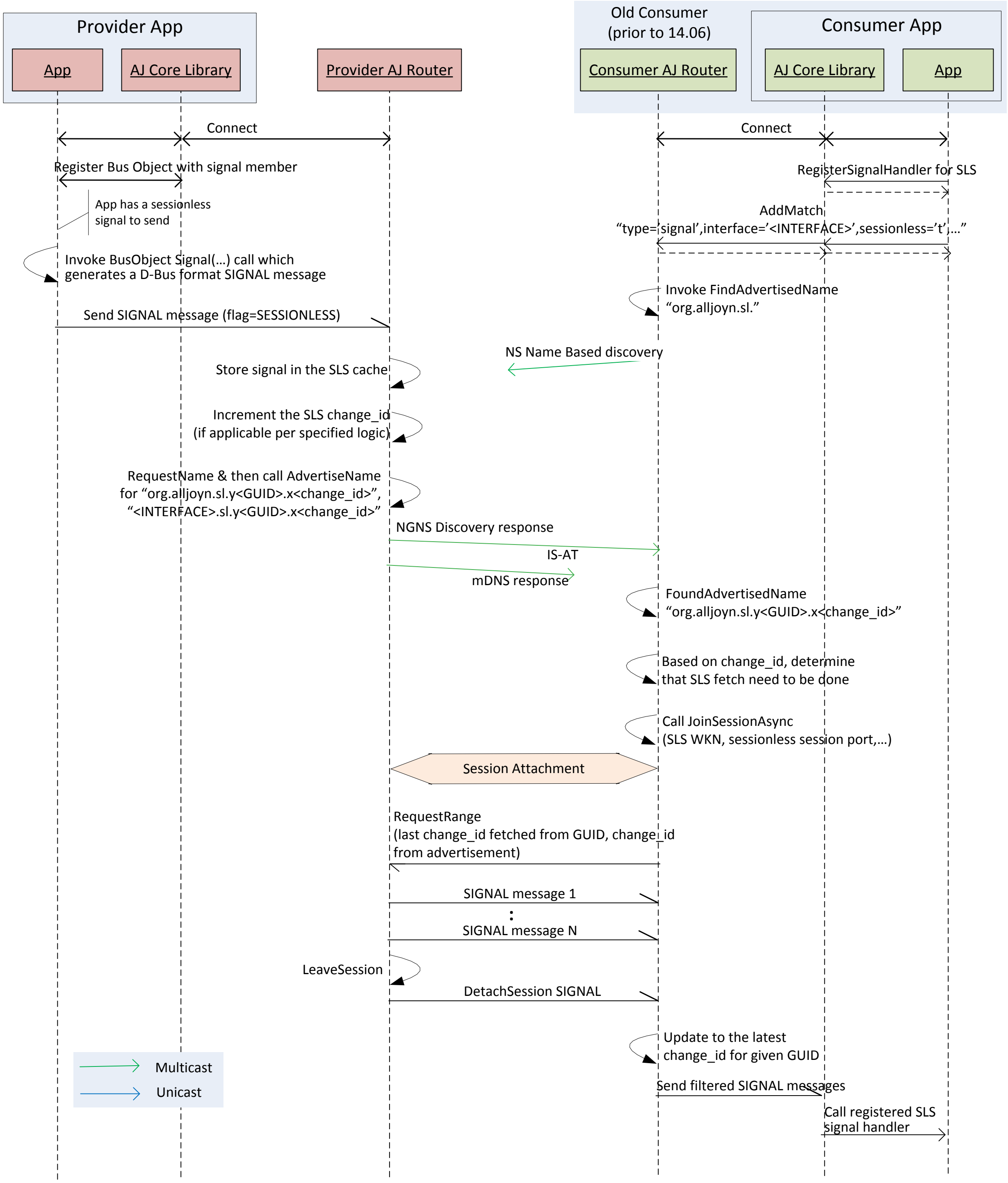

Provider App
App
AJ Core Library
Provider AJ Router
Old Consumer
(prior to 14.06)
Consumer AJ Router
Consumer App
AJ Core Library
App
Connect
Connect
Register Bus Object with signal member
RegisterSignalHandler for SLS
App has a sessionless
signal to send
AddMatch
“type=’signal’,interface=’<INTERFACE>’,sessionless=’t’,...”
Invoke BusObject Signal(...) call which
generates a D-Bus format SIGNAL message
Invoke FindAdvertisedName
“org.alljoyn.sl.”
Send SIGNAL message (flag=SESSIONLESS)
NS Name Based discovery
Store signal in the SLS cache
Increment the SLS change_id
(if applicable per specified logic)
RequestName & then call AdvertiseName
for “org.alljoyn.sl.y<GUID>.x<change_id>”,
“<INTERFACE>.sl.y<GUID>.x<change_id>”
NGNS Discovery response
IS-AT
mDNS response
FoundAdvertisedName
“org.alljoyn.sl.y<GUID>.x<change_id>”
Based on change_id, determine
that SLS fetch need to be done
Call JoinSessionAsync
(SLS WKN, sessionless session port,...)
Session Attachment
RequestRange
(last change_id fetched from GUID, change_id
from advertisement)
SIGNAL message 1
:
SIGNAL message N
LeaveSession
DetachSession SIGNAL
Update to the latest
change_id for given GUID
Multicast
Unicast
Send filtered SIGNAL messages
Call registered SLS
signal handler

Figure 7-12. SLS Logic between legacy consumer and new provider

7.4.4 Subsequent AddMatch done by an app

After the first AddMatch call to the AllJoyn router, applications can subsequently invoke other AddMatch calls to add more match rules for sessionless signals.

Starting from the 14.06 release, whenever an AddMatch call is invoked by any app, the SLS module takes steps to trigger fetch for sessionless signal messages. The logic is quite similar to the first sessionless signal delivery use case in section 7.4.1. The additional step is to fetch sessionless signal messages from existing providers that match the specified filtering criteria for sessionless signals.

The consumer SLS module remembers the sessionless signal advertised name it has received from each provider. For the new AddMatch rule that doesn't include implements key, the consumer uses the match rule to find matching providers among existing providers and fetches sessionless signals from those providers.

The consumer SLS module performs the following steps.

1. Perform interface name discovery for providers for any new 'implements' key/value pair specified in AddMatch. Fetch the sessionless signals from the discovered providers.
2. Perform name-based discovery for "<INTERFACE>.sl" for any new interface value specified in AddMatch, for which discovery was not already done. Fetch the sessionless signals from the discovered providers.
3. Fetch the sessionless signals from the already known providers if the sessionless signal name received from these providers match the filtering criteria specified in match rule.

7.5 org.alljoyn.sl interface

The org.alljoyn.sl interface is the AllJoyn interface between two AllJoyn routers used to enable the exchange of sessionless signals. Table 7-1 lists the org.alljoyn.sl interface signals.

Table 7-1. org.alljoyn.sl interface signals

Signal name	Parameters		Description
	Parameter name	Description	
RequestSignals	UINT32 fromId	Start of change_id range.	Requests sessionless signals associated with change_ids in the range [fromId, currentChangeId], where currentChangeId is the most recently advertised change_id of the provider.

Signal name	**Parameters**		**Description**
	Parameter name	**Description**	
RequestRange	UINT32 fromId	Start of change_id range	A signal for requesting sessionless signals associated with change_ids in the range [fromId, toId). **NOTE:** The “toId” is exclusive so a consumer should set toId=<change_id_value>+1 if it wants to get SLS up to the change_id_value. This signal appeared in version 6 of the AllJoyn protocol.
	UINT32 toId	End of change_id range	
RequestRangeMatch	UINT32 fromId	Start of change_id range	A signal for requesting sessionless signals associated with change_ids in the range [fromId, toId) that match any of the provided match rules. The “toId” is exclusive in this signal too. This signal appeared in version 10 of the AllJoyn protocol (associated with the 14.06 release).
	UINT32 toId	End of change_id range	
	ARRAY of STRING matchRules	Match rules to apply to the range	

8 AllJoyn Security

8.1 Overview

The AllJoyn system provides a security framework for applications to authenticate each other and send encrypted data between them. The AllJoyn framework provides end-to-end application level security. Authentication and data encryption are done at the application. These applications can reside on the same device or on different devices, and can be attached to the same AllJoyn router or different AllJoyn routers.

NOTE

No authentication is done at the AllJoyn router layer.

The AllJoyn framework supports security at the interface level. An application can tag an interface as ‘secure’ to enable authentication and encryption. All of the methods, signals, and properties of a secure interface are considered secure. Authentication- and encryption-related key exchange are initiated on demand when a consumer application invokes a method call on a secure interface, or explicitly invokes an API to secure the connection with a remote peer application.

Figure 8-1 shows the high-level AllJoyn security architecture. Authentication and encryption is done at the application layer. The AllJoyn core library implements all of the logic for authentication and encryption except the Auth Listener. The Auth Listener is a callback function implemented by the application to provide auth credentials (e.g., PIN or password) or verify auth credentials (e.g., verify certificate chain in case of ALLJOYN_RSA_KEYX). Authentication and encryption keys are stored in a key store managed by the Security module.

NOTE

The AllJoyn router is only involved in transmitting security-related messages between application endpoints. It does not implement any security logic itself.

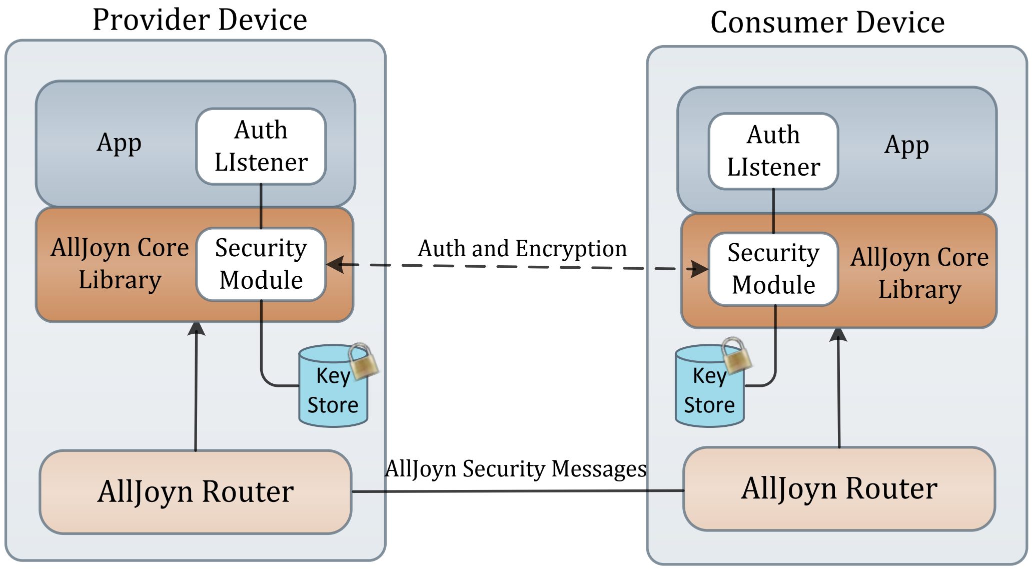


Figure 8-1. AllJoyn Security architecture

The AllJoyn framework uses the Simple Authentication and Security Layer (SASL) security framework for authentication. It makes use of D-Bus defined SASL protocol [1] for exchanging authentication related data.

The AllJoyn framework supports the following auth mechanisms for app-to-app level authentication:

- ALLJOYN_SRP_KEYX - Secure Remote Password (SRP) key exchange
- ALLJOYN _SRP_LOGON - Secure Remote Password (SRP) logon with username and password
- ALLJOYN _RSA_KEYX - RSA key-exchange using X509 certificates
- ALLJOYN _PIN_KEYX – PIN-based key-exchange developed for thin applications.

The AllJoyn framework also supports ANONYMOUS and EXTERNAL auth mechanisms as defined by the D-Bus specification.

- The ANONYMOUS auth mechanism is used between two AllJoyn routers for null authentication. It is also used for authentication between a thin app and an AllJoyn router.
- The EXTERNAL auth mechanism is used between an application and the installed AllJoyn router (standalone AllJoyn router) on the Linux platform.

8.1.1 Security changes in the 14.06 release

In the 14.06 release, the ALLJOYN _PIN_KEYX auth mechanism is removed from the AllJoyn thin core library. This auth mechanism continues to be supported by the AllJoyn standard core library.

The following new Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) based auth mechanism are added:

- ECDHE_NULL is an anonymous key agreement. There is no PIN or passphrase required.
- ECDHE_PSK is a key agreement authenticated with a pre-shared key like a PIN, passphrase, or symmetric key.
- ECDHE_ECDSA is a key agreement authenticated with an asymmetric key validated with an ECDSA signature.

These new auth mechanisms can be used by both thin apps and standard apps. Thin apps in the 14.06 release support only ECDHE-based auth mechanisms.

Use of SASL protocol for authentication is removed from the AllJoyn thin core library in the 14.06 release, and will continue to be supported in AllJoyn standard core library.

For more information about these changes, refer to [4].

8.2 Security concepts

This section defines the AllJoyn security-related concepts.

8.2.1 Authentication (Auth) GUID

The Authentication GUID is a GUID assigned to an application for authentication purposes. This GUID is persisted in the key store and provides a long-term identity for the application. Typically, this GUID is associated with a single application. In the scenario where a group of related applications share a given key store, they also share the same auth GUID.

This GUID is used as a mapping key for storing and accessing authentication and encryption keys for the associated application.

8.2.2 Master secret

The master secret is a key shared between authenticated peer applications. Two peer applications generate the same master secret independently, and store it persistently in the key store.

The master secret is stored per Auth GUID for peer applications, and has an associated TTL settable by the application. As long as the master secret is valid, peer applications do not have to reauthenticate with each other to exchange encrypted data.

The master secret is 48 bytes in length as per [6].

8.2.3 Session key

A cryptographic key used to encrypt point-to-point data traffic between two peer applications. A separate session key is maintained for every connected peer application. A session key is valid as long as peers are connected (over any AllJoyn session). It is a session in the cryptographic sense and not related to an AllJoyn session. The same session key is used across all AllJoyn sessions between two peers.

Session keys are stored in the memory, they are not persisted. The session key is generated after a peer application has been authenticated, and it expires for a peer application when the connection is terminated with that peer. The session key is derived from the master secret, and is used to encrypt method calls, method replies, and unicast signals.

The session key is 128 bits long.

NOTE

The current implementation has a default TTL of 2 days for session keys. If applications remain connected that long, the associated session key expires and a new session key would need to be generated.

8.2.4 Group key

The group key is a cryptographic key used to encrypt point-to-multipoint data traffic (broadcast signals) sent out by a provider application. A single group key is maintained by an application to encrypt broadcast signals sent to every connected peer application.

A group key is generated when an application generates the very first session key for any connected peer. The group key is always generated independent of the provider or consumer role of an application. Only provider applications use the group key to send out encrypted broadcast signals. Applications exchange their group keys using an encrypted method call that involves the session key.

The group key is 128 bits long and randomly generated. The group key is directional in nature. Each application has its own group key to encrypt broadcast signals. In addition, it also maintain a separate peer group key for each of the connected peer for decrypting broadcast signals received from them.

Group keys are stored in the memory, they are not persisted. The group expires when the connection ends with peer applications. An application’s own group key will expire when it no longer has any connections with any of its peers. The group key for a remote peer will expire when the application no longer has a connection with that peer.

NOTE

In future releases, group keys may be persisted to support encryption for sessionless signals.

8.2.5 Key store

The key store is a local storage used to persistently store authentication-related keys, and to store master secret and associated TTL. Applications can provide their own implementation of the key store or use the default key store provided by the AllJoyn system.

Multiple applications on a device can share a given key store. In this case, those applications share the same set of authentication keys. In the current implementation, content inside the key store is encrypted with a key derived from the key store path.

For every authenticated application, the key store maintains the master secret and the associated TTL per Auth GUID of that application. It also maintains the auth GUID assigned to the local applications that are using the key store.

Table 8-1 shows an example key store with master secret stored for two peer applications.

Table 8-1. Key store example with master secret stored for two peer applications

Local Auth GUID – GUIDx		
Peer Auth GUID	**Master Secret**	**TTL**
GUID1	MS1	T1
GUID2	MS2	T2

8.3 End-to-end security flow

Figure 8-2 captures the high-level end-to-end message flow for AllJoyn security for the use case when two applications have not authenticated with each other before. The security message flow is initiated based on one of the following triggers:

- Consumer app invoking a secure method call on a remote service object or
- Consumer app explicitly invoking an API to secure the connection with the remote peer

It is ideal for an application to always explicitly secure the connection with the remote peer. In the case when an application is just interested in receiving secure signals, that is the only way to secure the connection with the remote peer in order to receive keys for decrypting signals.

The AllJoyn core library attached with the application implements all of the AllJoyn security logic. The AllJoyn router only acts as a pass-through for security-related messages. Each application needs to invoke EnablePeerSecurity API call with the AllJoyn core library to enable AllJoyn security. The application specifies authentication mechanism to be used, the Auth Listener for callback and the key store file as part of this API call. It also indicates whether key store can be shared. The AllJoyn core library generates the auth GUID for the application as part of first-time initialization of the key store. The auth GUID gets stored in the key store.

After establishing a session with the provider app, consumer app initiates one of the security triggers mentioned above. The AllJoyn Core library checks to see if authentication is already in progress.

- If yes, it stops.
- If no, it continues with the security flow.

It looks for the key material for remote peer. In this case, since this is the first secure interaction with the remote app, no key material is found. This will trigger security flow with the remote peer.

The message flow consists of following four distinct steps in that order:

1. **Exchange Auth GUIDs:** This step involves exchanging Auth GUIDs between peer applications. Once learned, the remote app auth GUID is used to see if the master secret is present for that auth GUID in the key store. In this case, no master secret is found since the two apps have not authenticated with each other.
2. **App-to-App authentication:** This step involves two peers authenticating each other using one of the supported auth mechanisms. At the end of this step, two peers have authenticated each other and now share a common master key.
3. **Generate a session key:** This step involves two peers generating a session key to be used for encrypting secure point-to-point messages between them. The session key is generated independently by both the peers based on the shared master key. A group key is also generated when the first session key is generated.
4. **Exchange group keys:** This step involves two peers exchanging their own group keys with each other via an encrypted AllJoyn message. The AllJoyn message gets encrypted using the session key already established between two peers. The group key is used by the application to encrypt session multicast and broadcast signals. At the end of this step, two peer applications have group keys to decrypt secure broadcast signals received from each other.

Details for each of these steps are described in the following sections. After completing these steps, peer applications have now established encryption/decryption keys to exchange encrypted method calls and signals.

These keys are managed as part of a peer state table which includes a unique name for the remote peer, as well as a local auth GUID and group key for the current application.

Table 8-2 provides a sample peer state table with keys stored for two authenticated peer applications.

Table 8-2. Sample peer state table for two authenticated peer applications

Local Auth GUID – GUIDx **App Group Key - GKx**			
Peer Auth GUID	**Unique Name**	**Session Key**	**Peer Group Key**
GUID1	:100.2	SK1	GK1
GUID2	:200.2	SK2	GK2

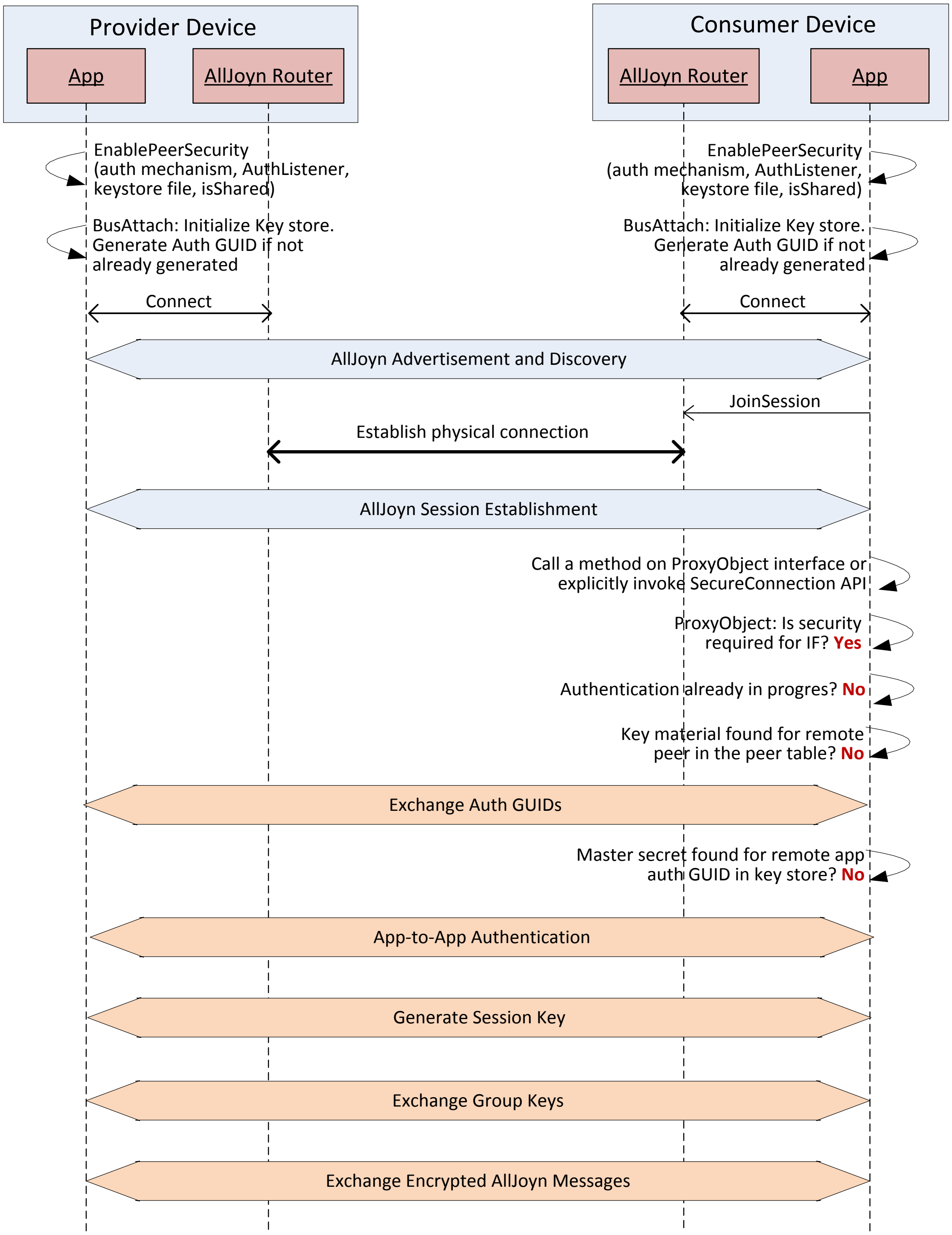

Provider Device
App
AllJoyn Router
Consumer Device
AllJoyn Router
App
EnablePeerSecurity
(auth mechanism, AuthListener,
keystore file, isShared)
BusAttach: Initialize Key store.
Generate Auth GUID if not
already generated
Connect
EnablePeerSecurity
(auth mechanism, AuthListener,
keystore file, isShared)
BusAttach: Initialize Key store.
Generate Auth GUID if not
already generated
Connect
AllJoyn Advertisement and Discovery
JoinSession
Establish physical connection
AllJoyn Session Establishment
Call a method on ProxyObject interface or
explicitly invoke SecureConnection API
ProxyObject: Is security
required for IF? Yes
Authentication already in progres? No
Key material found for remote
peer in the peer table? No
Exchange Auth GUIDs
Master secret found for remote app
auth GUID in key store? No
App-to-App Authentication
Generate Session Key
Exchange Group Keys
Exchange Encrypted AllJoyn Messages

Figure 8-2. End-to-end security flow (two applications have not authenticated with each other before)

8.4 Already authenticated apps

When apps connect with each other subsequent to their first connection, they do not need to authenticate again with each other if the master secret is still valid. Figure 8-3 shows security flow for this use case.

In this case, since the apps were not connected, no key material is found for the remote peer. As a result, the consumer app performs the Exchange Auth GUIDs step with the remote peer. This retrieves the Auth GUID for the remote peer that can be used for lookup in the key store file.

Since the apps have already authenticated, a master secret is found in the key store for the auth GUID of the provider app and no app-to-app authentication needs to occur. The consumer app directly goes to the next step of generating session key and/or group key.

NOTE

If the verification step fails during session key generation, the consumer app must re-authenticate with the provider even if master secret is still valid.

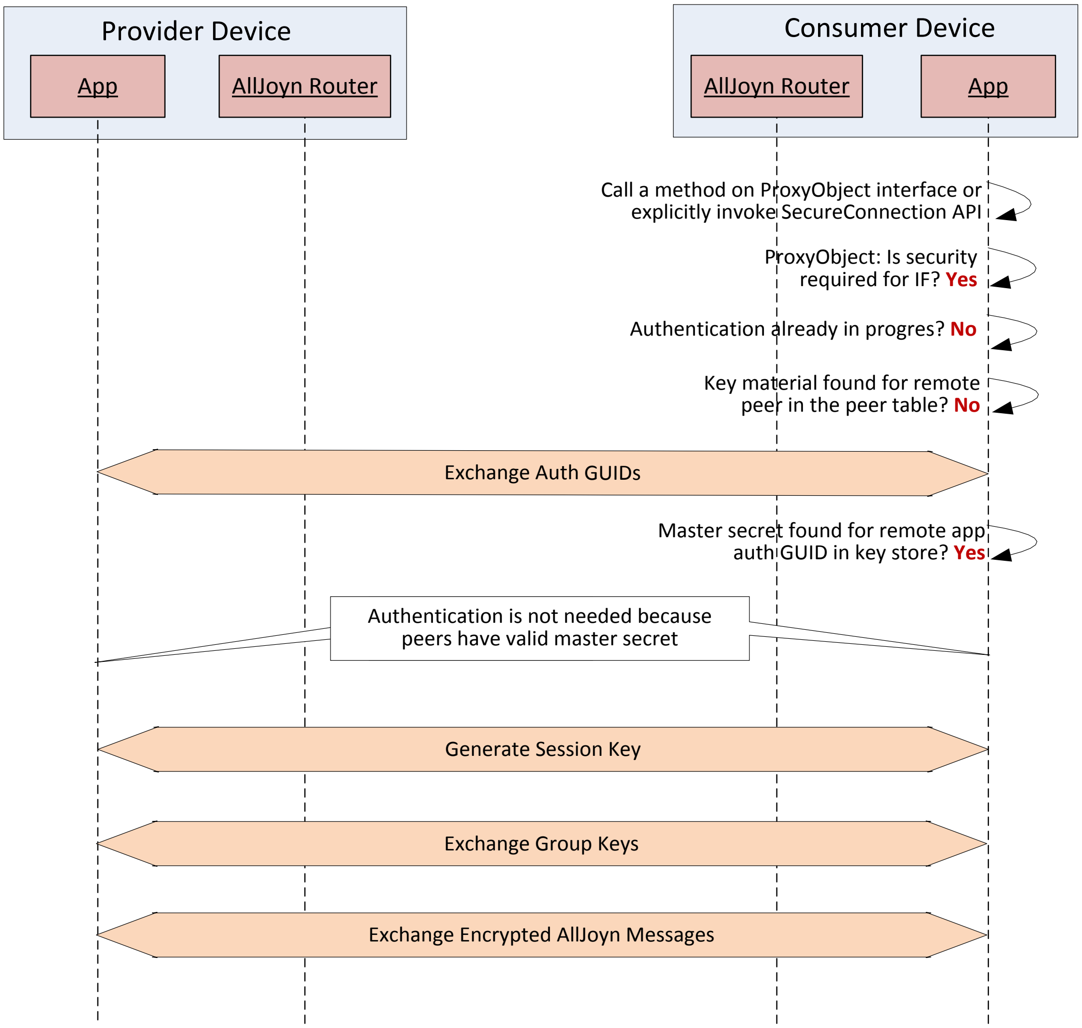

Provider Device
App
AllJoyn Router
Consumer Device
AllJoyn Router
App
Call a method on ProxyObject interface or explicitly invoke SecureConnection API
ProxyObject: Is security required for IF? Yes
Authentication already in progres? No
Key material found for remote peer in the peer table? No
Exchange Auth GUIDs
Master secret found for remote app auth GUID in key store? Yes
Authentication is not needed because peers have valid master secret
Generate Session Key
Exchange Group Keys
Exchange Encrypted AllJoyn Messages

Figure 8-3. Authenticated apps connecting again

8.5 Exchange of Auth GUIDs

Figure 8-4 shows the message flow for the exchange of Auth GUIDs between peer applications.

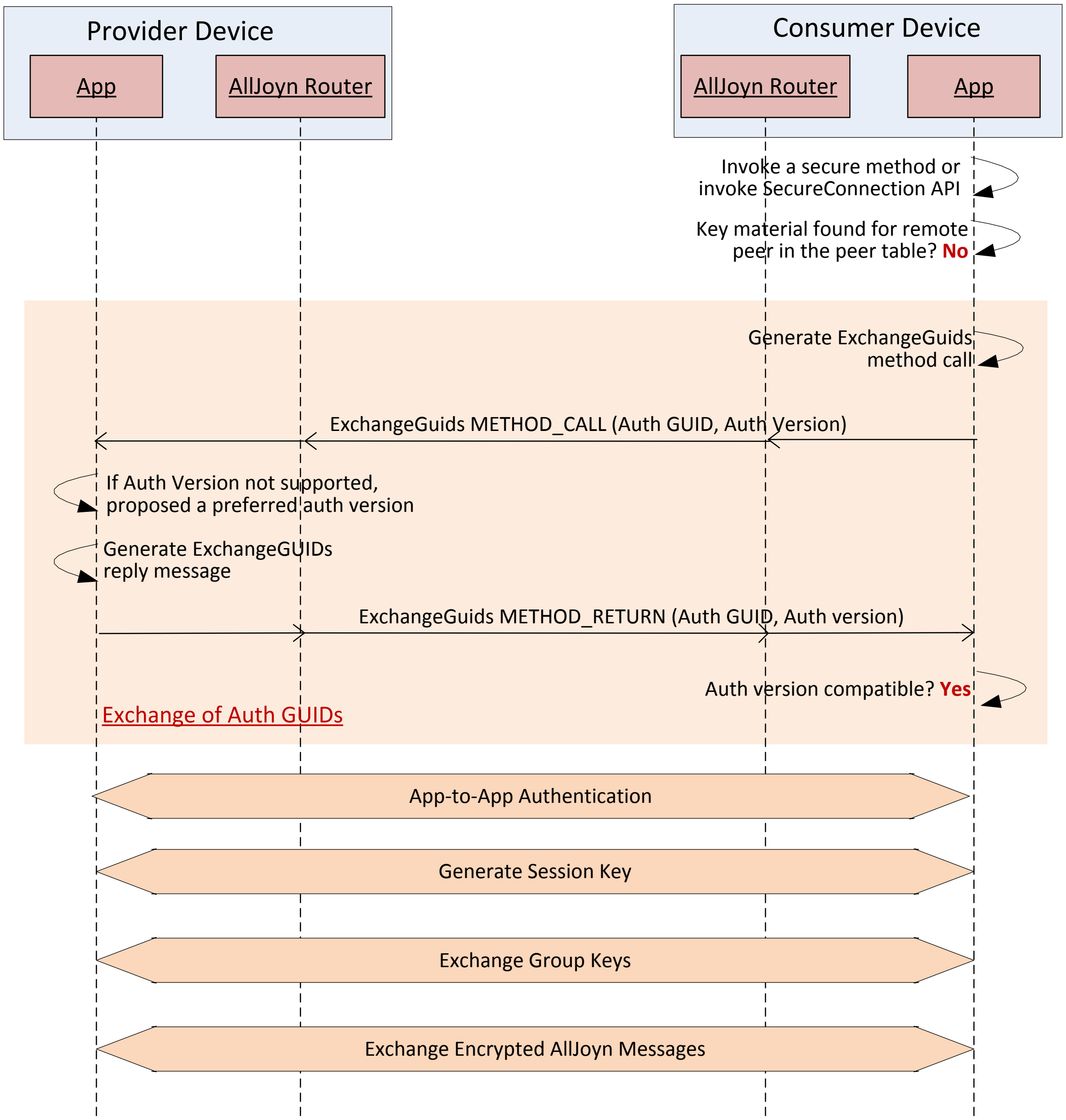

Provider Device
App
AllJoyn Router
Consumer Device
AllJoyn Router
App
Invoke a secure method or invoke SecureConnection API
Key material found for remote peer in the peer table? No
Generate ExchangeGuids method call
ExchangeGuids METHOD_CALL (Auth GUID, Auth Version)
If Auth Version not supported, proposed a preferred auth version
Generate ExchangeGUIDs reply message
ExchangeGuids METHOD_RETURN (Auth GUID, Auth version)
Auth version compatible? Yes
Exchange of Auth GUIDs
App-to-App Authentication
Generate Session Key
Exchange Group Keys
Exchange Encrypted AllJoyn Messages

Figure 8-4. Exchange of Auth GUIDs

The message flow steps are described below.

1. The consumer app generates an ExchangeGuids METHOD_CALL message and sends it to the provider app via the AllJoyn router. This message includes the Auth GUID of the consumer app and the maximum Auth version supported by the consumer app.
2. The provider app proposes its max auth version if it does not support the auth version received from the consumer app.

3. The provider app generates an ExchangeGuids METHOD_RETURN message and sends it to the consumer app via the AllJoyn router. This message includes the Auth GUID of the provider app and the max auth version of the provider app.
4. The consumer app verifies that it supports the received auth version. This completes the Exchange GUIDs step.

8.6 App-to-app authentication

AllJoyn peer applications authenticate each other using one of the auth mechanisms detailed in this section. These auth mechanisms are designed based on the security constructs in [6] and [7]. Applicable RFC sections are listed when describing details for these auth mechanisms.

NOTE

For the authentication message flows captured in this section, the consumer and provider apps are also referred to as client and server respectively, to correspond with terminology used in [6] and [7].

8.6.1 Use of D-Bus SASL protocol

The AllJoyn framework implements the D-Bus SASL exchange protocol [1] to exchange authentication-related data. All authentication-related exchanges are done using the AuthChallenge method call/reply defined as part of the org.alljoyn.Bus.Peer.Authentication interface implemented by the AllJoyn core library.

Auth data to be exchanged is generated as a SASL string based on the D-Bus SASL exchange protocol. Auth data inside the SASL string is sent in the hex form. The generated string is then passed as a parameter to the AuthChallenge method call or method reply.

For example, to initiate authentication for ALLJOYN_RSA_KEYX, generated string would be:

“AUTH ALLJOYN_RSA_KEYX <c_rand in hex>”

This includes the SASL AUTH command, auth mechanism, and auth data in hex form.

Table 8-3 captures the D-Bus SASL commands supported by the AllJoyn framework.

Table 8-3. D-Bus SASL commands supported by the AllJoyn framework

Command	Direction	Description
AUTH [mechanism] [initial-response]	Consumer→Provider	Start the authentication.
CANCEL	Consumer→Provider	Cancel the authentication

Command	Direction	Description
BEGIN	Consumer→Provider Provider→Consumer	■ On the consumer side, acknowledge that the consumer has received an OK command from the provider, and that the stream of messages is about to begin. ■ From the provider side, sent by the provider as response to BEGIN command from the consumer.
DATA	Consumer→Provider Provider→Consumer	On the consumer or provider side, contains a hex-encoded block of data to be interpreted according to the auth mechanism in use.
OK	Consumer→Provider	The client has been authenticated.
REJECTED	Consumer→Provider	On the consumer side, indicates that the current authentication exchange has failed, and further exchange of DATA is inappropriate. The consumer tries another mechanism, or tries providing different responses to challenges.
ERROR	Consumer→Provider Provider→Consumer	On the consumer or provider side, either the provider or consumer did not know a command, does not accept the given command in the current context, or did not understand the arguments to the command.

8.6.2 ALLJOYN _RSA_KEYX

The ALLJOYN_RSA_KEYX auth mechanism is designed to be primarily used by applications that can present certificates issued by a common CA to each other for authentication purpose. Implementation supports self-signed certificates as well.

Figure 8-5 shows the message flow for the ALLJOYN_RSA_KEYX auth mechanism. Both sides maintain a running hash of security messages exchanged which is used by the server to verify that it is talking to the same client to minimize man-in-the middle attack.

NOTE

It was decided to not include any certificate chain verification inside the AllJoyn Security implementation because each platform provides platform-specific mechanism to verify the certificate chain.

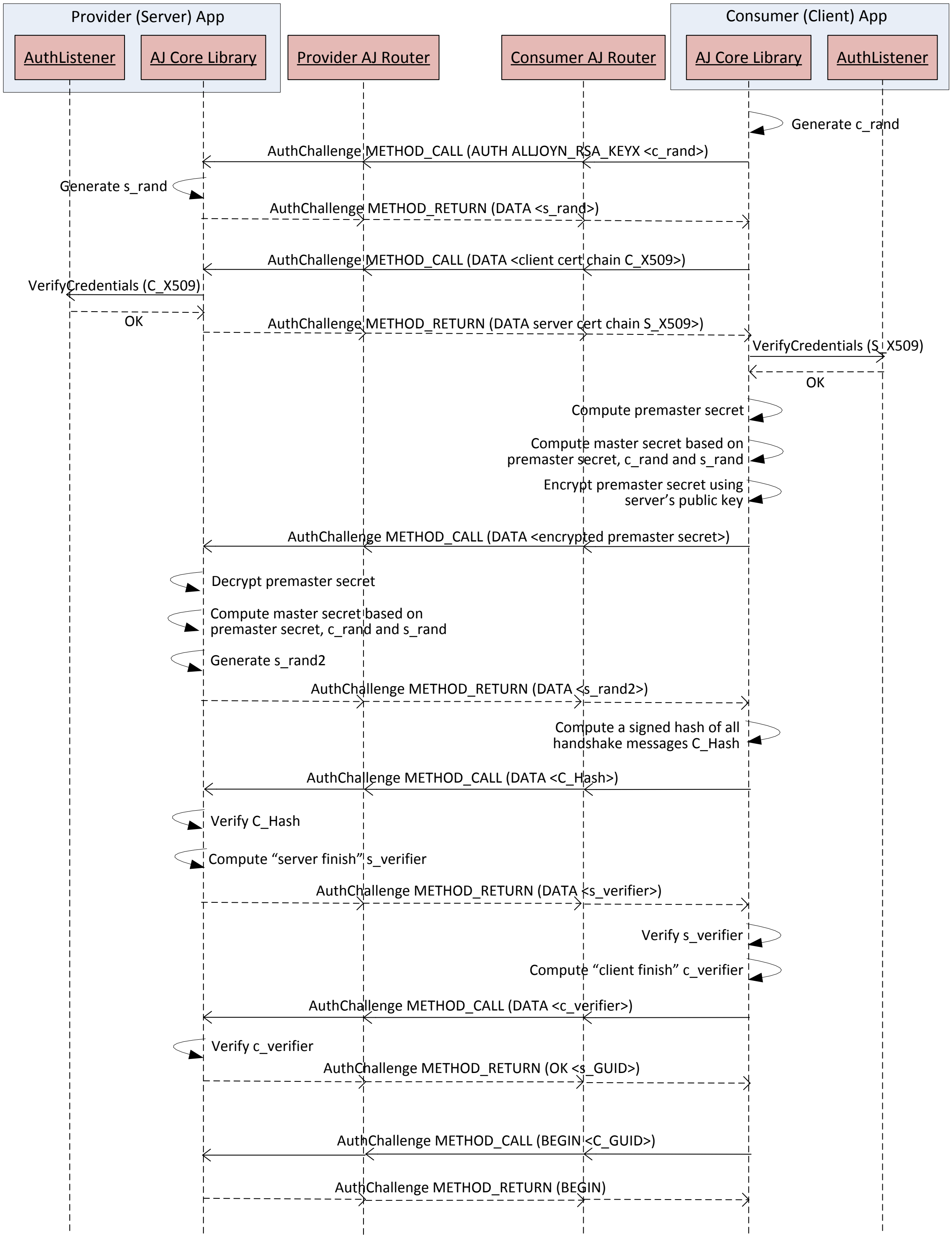

Provider (Server) App
AuthListener
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer (Client) App
AJ Core Library
AuthListener
Generate c_rand
AuthChallenge METHOD_CALL (AUTH ALLJOYN_RSA_KEYX <c_rand>)
Generate s_rand
AuthChallenge METHOD_RETURN (DATA <s_rand>)
AuthChallenge METHOD_CALL (DATA <client cert chain C_X509>)
VerifyCredentials (C_X509)
OK
AuthChallenge METHOD_RETURN (DATA server cert chain S_X509>)
VerifyCredentials (S_X509)
OK
Compute premaster secret
Compute master secret based on premaster secret, c_rand and s_rand
Encrypt premaster secret using server's public key
AuthChallenge METHOD_CALL (DATA <encrypted premaster secret>)
Decrypt premaster secret
Compute master secret based on premaster secret, c_rand and s_rand
Generate s_rand2
AuthChallenge METHOD_RETURN (DATA <s_rand2>)
Compute a signed hash of all handshake messages C_Hash
AuthChallenge METHOD_CALL (DATA <C_Hash>)
Verify C_Hash
Compute "server finish" s_verifier
AuthChallenge METHOD_RETURN (DATA <s_verifier>)
Verify s_verifier
Compute "client finish" c_verifier
AuthChallenge METHOD_CALL (DATA <c_verifier>)
Verify c_verifier
AuthChallenge METHOD_RETURN (OK <s_GUID>)
AuthChallenge METHOD_CALL (BEGIN <C_GUID>)
AuthChallenge METHOD_RETURN (BEGIN)

Figure 8-5. ALLJOYN _RSA_KEYX auth mechanism

The message flow steps are described below.

1. The consumer (client) app generates a 28 bytes client random string c_rand.
2. The consumer app generates an AuthChallenge METHOD_CALL message and passes “AUTH ALLJOYN_RSA_KEYX <c_rand>” as parameter in that message. The consumer app sends the method call to the provider (server) app via the AllJoyn router.
3. The provider app generates a 28 bytes server random string s_rand.
4. The provider app generates an AuthChallenge METHOD_RETURN message and passes “DATA <s_rand>” as parameter in that message. The provider app sends a method reply to the consumer app via the AllJoyn router.
5. The consumer app generates an AuthChallenge METHOD_CALL message to send the client’s certificate chain C_X509 to the server. \
   - The client certificate chain is specified using X.509 type. Refer to section 7.4.6 in [6].
   - The consumer app passes “DATA <C_X509>”as parameter to the method call.
   - The consumer app sends the method call to the provider (server) app via the AllJoyn router.
6. The provider app invokes the AuthListener callback registered by the application to verify the client certificate chain C_X509. The AuthListener returns an OK response meaning the cert verification was successful.
7. The provider app generates an AuthChallenge METHOD_RETURN message and provides its own server certificate chain S_X509 in that message.
8. The provider app passes “DATA <S_X509>” as parameter in that message. Refer to section 7.4.2 in [6]. The provider app sends method reply to the consumer app via the AllJoyn router.
9. The consumer app invokes the AuthListener callback registered by the application to verify server certificate chain S_X509. The AuthListener returns an OK response meaning cert verification was successful.
10. The consumer app computes a 48 bytes premaster secret random.
11. The consumer app computes the master secret based on premaster secret, c_rand, and s_rand. The master secret is a 48 bytes value and is computed using algorithm described in section 8.1 in [6].
12. The consumer app encrypts the premaster secret using server’s public key.
13. The consumer app generates an AuthChallenge METHOD_CALL message to send encrypted premaster secret to the server. The consumer app passes “DATA <encrypted premaster secret>”as parameter to the method call. The consumer app sends method call to the provider (server) app via the AllJoyn router.
14. The provider app decrypts the premaster secret using its private key. The provider app then computes the master secret based on the premaster secret, c_rand, and s_rand using same algorithm used by the client (per section 8.1 in [6]).

15. The provider app generates another 28 bytes random string s_rand2.
16. The provider app generates an AuthChallenge METHOD_RETURN message and passes “DATA <s_rand2>” as parameter in that message. The provider app sends method reply to the consumer app via the AllJoyn router.
17. The consumer app computes a signed hash C_Hash of all the handshake messages exchanged up to this point. The hash is signed with client’s private key. Refer to section 7.4.8 in [6].
18. The consumer app generates an AuthChallenge METHOD_CALL message to send the signed hash to the server. The consumer app passes “DATA <C_Hash>”as parameter to the method call. The consumer app sends the method call to the provider (server) app via the AllJoyn router.
19. The provider app decrypts the signed hash with client’s public key. It then verifies that the received hash value is the same as the hash of handshake messages it has maintained (excluding the last message).
20. The provider app computes a “server finish” s_verifier as per the algorithm in section 7.4.9 of [6]. The s_verifier is generated based on the master secret, hash of handshake messages, and “server finish” label.
21. The provider app generates an AuthChallenge METHOD_RETURN message and passes “DATA <s_verifier>” as parameter in that message. The provider app sends the method reply to the consumer app via the AllJoyn router.
22. The consumer app generates the “server finish” verifier using the same algorithm as the provider app, and verifies that the computed value is same as the received s_verifier.
23. The consumer app computes a “client finish” c_verifier as per the algorithm in section 7.4.9 of [6]. The c_verifier is generated based on the master secret, hash of handshake messages, and “client finish” label.
24. The consumer app generates an AuthChallenge METHOD_CALL message to send the c_verifier to the server. The consumer app passes “DATA <c_verifier>”as parameter to the method call. The consumer app sends method call to the provider (server) app via the AllJoyn router.
25. The provider app generates the “client finish” verifier using the same algorithm as the consumer app and verifies that the computed value is same as the received c_verifier.

    At this point, the client and server have authenticated with each other.
26. The provider app generates an AuthChallenge METHOD_RETURN message indicating authentication is complete. The provider app passes “OK <s_GUID>” as parameter in that message, where s_GUID is the auth GUID of the provider app. The provider app sends the method reply to the consumer app via the AllJoyn router.
27. The consumer app sends an AuthChallenge METHOD_CALL to the provider app specifying “BEGIN <c_GUID>” as parameter. This indicates to the provider that the client has received the OK message, and the stream of data messages is about to begin. The c_GUID is the auth GUID of the consumer app.

28. The provider app sends an AuthChallenge METHOD_RETURN message specifying “BEGIN” as parameter.

8.6.3 ALLJOYN_SRP_KEYX

Figure 8-6 shows the message flow for the ALLJOYN_SRP_KEYX auth mechanism. This auth mechanism is primarily designed for use cases where a one-time use password is generated by both sides.

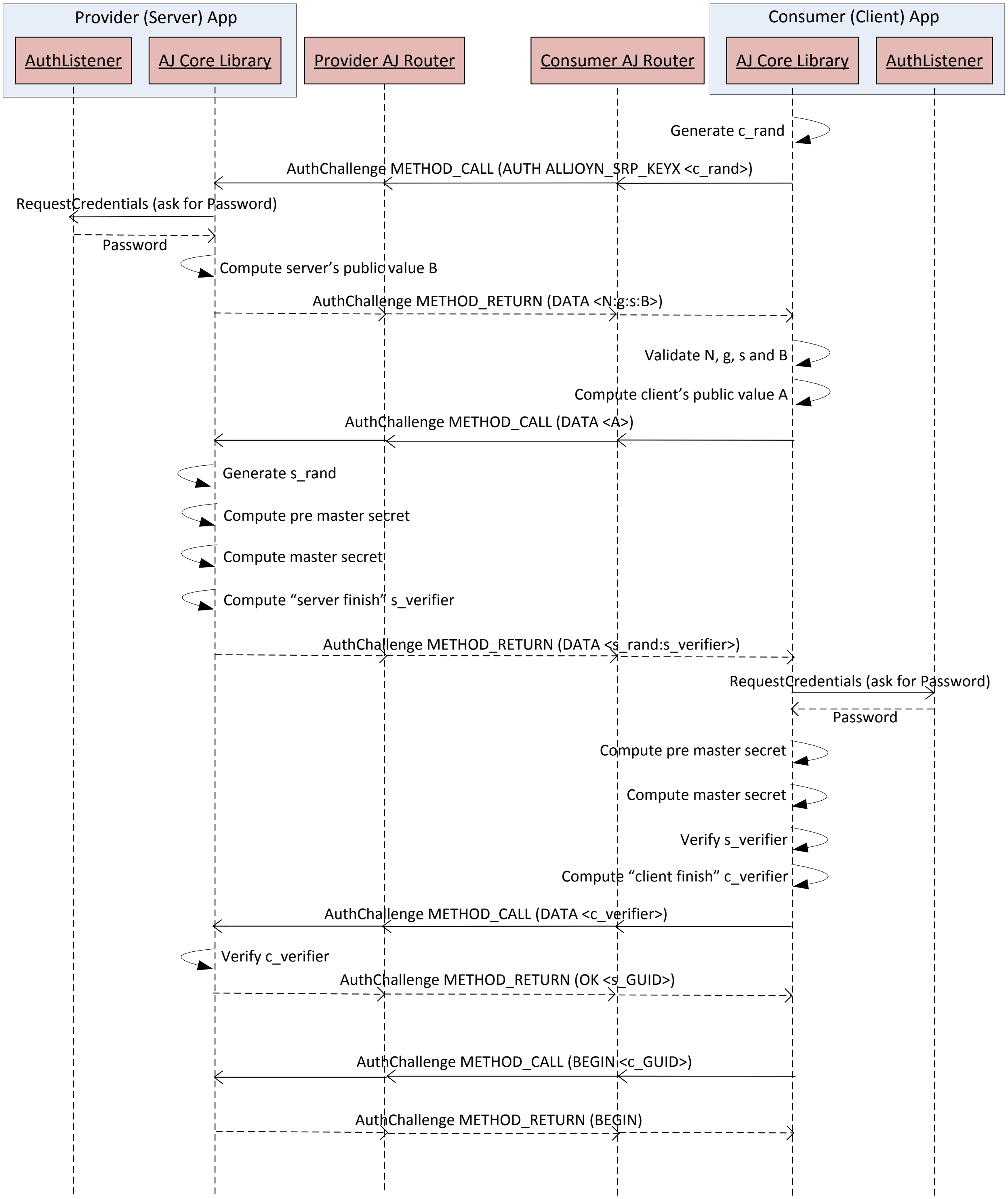

Provider (Server) App
AuthListener
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer (Client) App
AJ Core Library
AuthListener
Generate c_rand
AuthChallenge METHOD_CALL (AUTH ALLJOYN_SRP_KEYX <c_rand>)
RequestCredentials (ask for Password)
Password
Compute server's public value B
AuthChallenge METHOD_RETURN (DATA <N:g:s:B>)
Validate N, g, s and B
Compute client's public value A
AuthChallenge METHOD_CALL (DATA <A>)
Generate s_rand
Compute pre master secret
Compute master secret
Compute "server finish" s_verifier
AuthChallenge METHOD_RETURN (DATA <s_rand:s_verifier>)
RequestCredentials (ask for Password)
Password
Compute pre master secret
Compute master secret
Verify s_verifier
Compute "client finish" c_verifier
AuthChallenge METHOD_CALL (DATA <c_verifier>)
Verify c_verifier
AuthChallenge METHOD_RETURN (OK <s_GUID>)
AuthChallenge METHOD_CALL (BEGIN <c_GUID>)
AuthChallenge METHOD_RETURN (BEGIN)

Figure 8-6. ALLJOYN_SRP_KEYX auth mechanism

The message flow steps are described below.

1. The consumer app generates a 28 bytes client random string c_rand.
2. The consumer (client) app generates an AuthChallenge METHOD_CALL message and passes “AUTH ALLJOYN_SRP_KEYX <c_rand>” as parameter in that message. The consumer app sends the method call to the provider (server) app via the AllJoyn router.
3. The provider app invokes the AuthListener callback registered by the application to request for a password. The AuthListener returns the password. A username of “anonymous” is used in this case.
4. The provider app computes the server’s public value B as per the algorithm in section 2.5.3 of RFC 5054.
5. The provider app generates an AuthChallenge METHOD_RETURN message to send a server key exchange message to the client. The provider app passes “DATA <N:g:s:B>” as parameter to that message. Refer to section 2.5.3 of [7]. The ‘s’ is a 40 bytes random salt value. The provider app sends method reply to the consumer app via the AllJoyn router.
6. The consumer app validates the values of N, g, s and B per section 2.5.3 of [7].
7. The consumer app computes the client’s public value A per section 2.5.4 of [7].
8. The consumer (client) app generates an AuthChallenge METHOD_CALL message and passes “DATA <A>” as parameter in that message. The consumer app sends the method call to the provider (server) app via the AllJoyn router.
9. The provider app generates a 28 bytes server random string s_rand.
10. The provider app computes a premaster secret using the algorithm in section 2.6 of RFC 5054. The premaster secret is based on the client’s public value (A), the server’s public value (B), and password among other parameters.
11. The provider app computes a master secret based on the premaster secret, c_rand, and s_rand as per the algorithm in section 8.1 of [6].
12. The provider app computes a “server finish” s_verifier as per the algorithm in section 7.4.9 of RFC 5246. The s_verifier is generated based on master secret, hash of handshake messages, and “server finish” label.
13. The provider app generates an AuthChallenge METHOD_RETURN message and passes “DATA <s_rand:s_verfier>” as parameter to that message. The provider app sends the method reply to the consumer app via the AllJoyn router.
14. The consumer app invokes the AuthListener callback registered by the application to request for a password. The AuthListener returns the password. A username of “anonymous” is used in this case.
15. The consumer app computes a premaster secret using the algorithm in section 2.6 of RFC 5054. The premaster secret is based on the client’s public value (A), the server’s public value (B), and the password among other parameters.
16. The consumer app computes a master secret based on the premaster secret, c_rand, and s_rand as per the algorithm in section 8.1 of [6].

17. The consumer app generates the “server finish” verifier using the same algorithm as the provider app and verifies that the computed value is same as the received s_verifier.
18. The consumer app computes a “client finish” c_verifier as per the algorithm in section 7.4.9 of RFC 5246. The c_verifier is generated based on the master secret, hash of handshake messages, and “client finish” label.
19. The consumer app generates an AuthChallenge METHOD_CALL message to send the c_verifier to the server. The consumer app passes “DATA <c_verifier>”as parameter to the method call. The consumer app sends the method call to the provider (server) app via the AllJoyn router.
20. The provider app generates the “client finish” verifier using the same algorithm as the consumer app and verifies that the computed value is same as the received c_verifier.

    At this point, the client and server have authenticated with each other.
21. The provider app generates an AuthChallenge METHOD_RETURN message indicating that authentication is complete. The provider app passes “OK <s_GUID>” as parameter in that message, where s_GUID is the auth GUID of the provider app. The provider app sends the method reply to the consumer app via the AllJoyn router.
22. The consumer app sends an AuthChallenge METHOD_CALL to the provider app specifying “BEGIN <c_GUID>” as parameter. This indicates to the provider that the client has received the OK message, and the stream of data messages is about to begin. The c_GUID is auth GUID of the consumer app.
23. The provider app sends an AuthChallenge METHOD_RETURN message, specifying “BEGIN” as parameter.

8.6.4 ALLJOYN_SRP_LOGON

Figure 8-7 shows the message flow for the ALLJOYN_SRP_LOGON auth mechanism. This auth mechanism is designed for client-server use cases where server maintains username and password, and the client uses this information for authentication. This mechanism is quite similar to the AllJoyn_SRP_KEYX auth mechanism with the following differences:

- The consumer app invokes the AuthListener callback up front to request the username and password from the application. The consumer app then passes the username in the first AuthChallenge message sent to the provider app.
- The provider app uses the received username to request for password from the AuthListener.

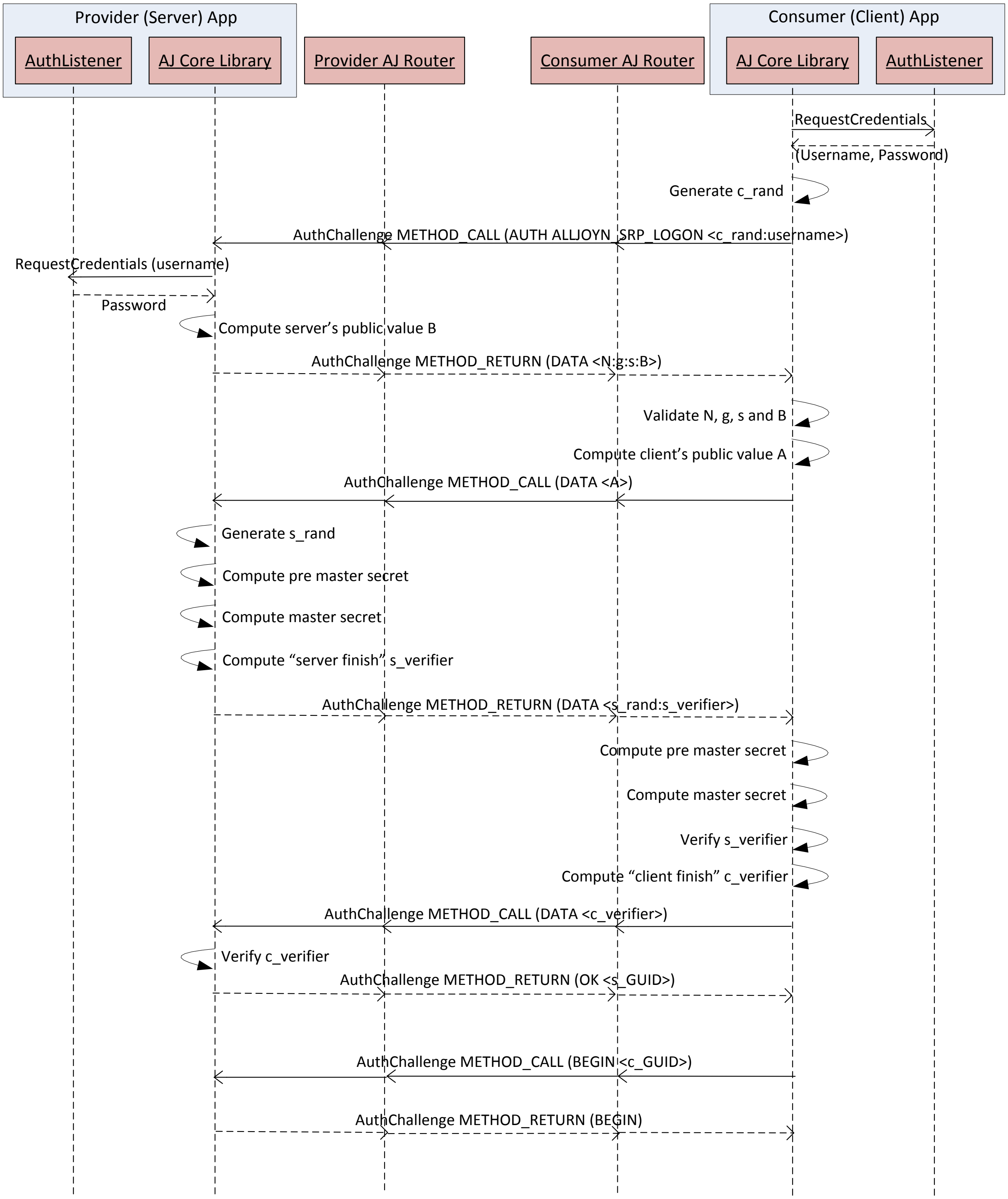

Provider (Server) App
AuthListener
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer (Client) App
AJ Core Library
AuthListener
RequestCredentials
(Username, Password)
Generate c_rand
AuthChallenge METHOD_CALL (AUTH ALLJOYN_SRP_LOGON <c_rand:username>)
RequestCredentials (username)
Password
Compute server's public value B
AuthChallenge METHOD_RETURN (DATA <N:g:s:B>)
Validate N, g, s and B
Compute client's public value A
AuthChallenge METHOD_CALL (DATA <A>)
Generate s_rand
Compute pre master secret
Compute master secret
Compute "server finish" s_verifier
AuthChallenge METHOD_RETURN (DATA <s_rand:s_verifier>)
Compute pre master secret
Compute master secret
Verify s_verifier
Compute "client finish" c_verifier
AuthChallenge METHOD_CALL (DATA <c_verifier>)
Verify c_verifier
AuthChallenge METHOD_RETURN (OK <s_GUID>)
AuthChallenge METHOD_CALL (BEGIN <c_GUID>)
AuthChallenge METHOD_RETURN (BEGIN)

Figure 8-7. ALLJOYN_SRP_LOGON auth mechanism

8.6.5 ALLJOYN_PIN_KEYX

Figure 8-8 shows the message flow for the ALLJOYN_PIN_KEYX auth mechanism. This auth mechanism is specifically designed for thin applications prior to the 14.06 release.

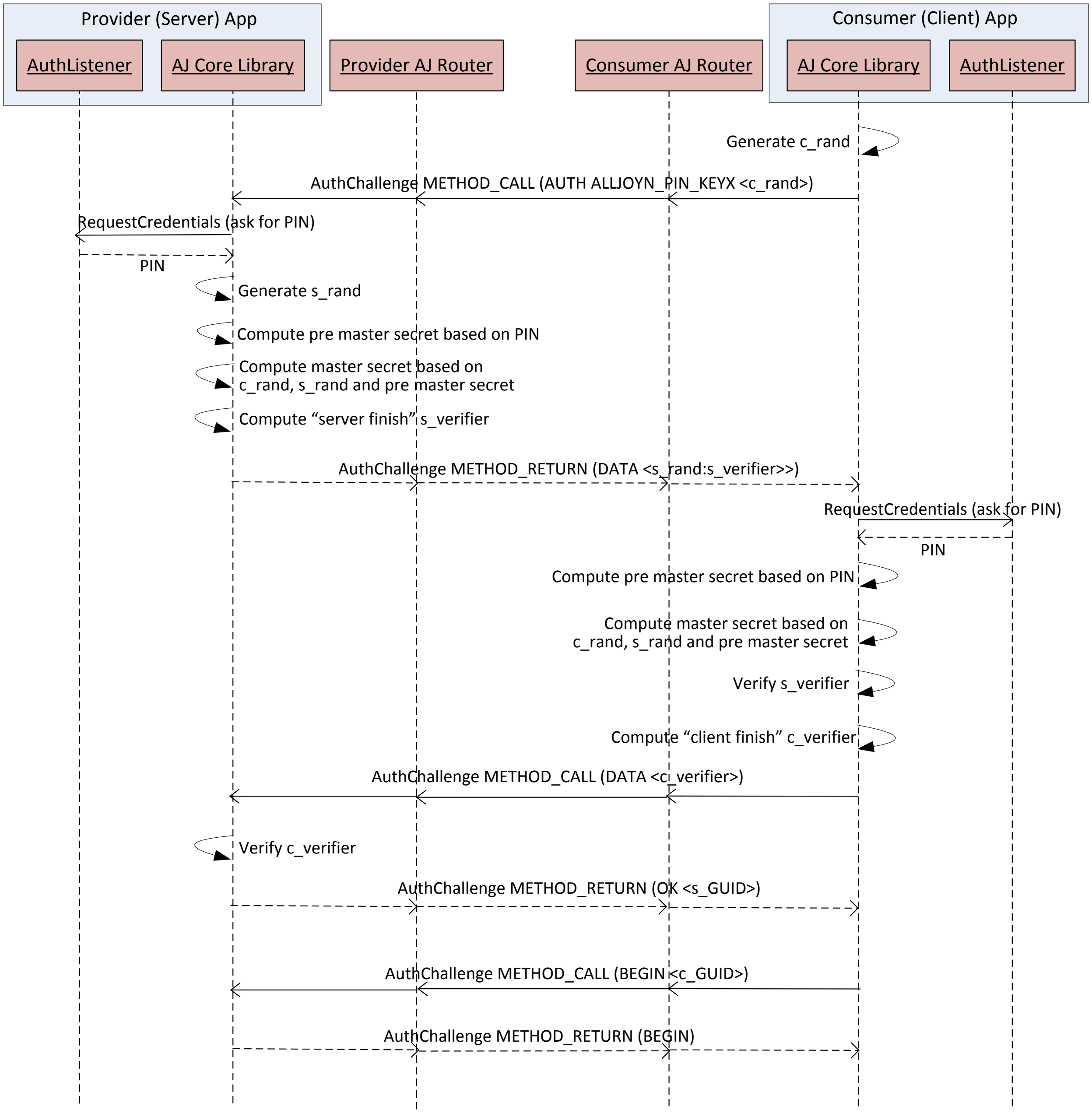

Provider (Server) App
AuthListener
AJ Core Library
Provider AJ Router
Consumer AJ Router
Consumer (Client) App
AJ Core Library
AuthListener
Generate c_rand
AuthChallenge METHOD_CALL (AUTH ALLJOYN_PIN_KEYX <c_rand>)
RequestCredentials (ask for PIN)
PIN
Generate s_rand
Compute pre master secret based on PIN
Compute master secret based on c_rand, s_rand and pre master secret
Compute “server finish” s_verifier
AuthChallenge METHOD_RETURN (DATA <s_rand:s_verifier>>)
RequestCredentials (ask for PIN)
PIN
Compute pre master secret based on PIN
Compute master secret based on c_rand, s_rand and pre master secret
Verify s_verifier
Compute “client finish” c_verifier
AuthChallenge METHOD_CALL (DATA <c_verifier>)
Verify c_verifier
AuthChallenge METHOD_RETURN (OK <s_GUID>)
AuthChallenge METHOD_CALL (BEGIN <c_GUID>)
AuthChallenge METHOD_RETURN (BEGIN)

Figure 8-8. ALLJOYN_PIN_KEYX auth mechanism

The message flow steps are described below.

1. The consumer app generates a 28 bytes client random string c_rand.

2. The consumer (client) app generates an AuthChallenge METHOD_CALL message and passes “AUTH ALLJOYN_PIN_KEYX <c_rand>” as parameter in that message. The consumer app sends the method call to the provider (server) app via the AllJoyn router.
3. The provider app invokes the AuthListener callback registered by the application to request a PIN. The AuthListener returns the PIN.
4. The provider app generates a 28 bytes server random string s_rand.
5. The provider app computes a premaster secret based on the PIN.
6. The provider app computes a master secret based on the premaster secret, c_rand, and s_rand as per the algorithm in section 8.1 of RFC 5246.
7. The provider app computes a “server finish” 16 bytes s_verifier using the AES CCM algorithm. The s_verifier is generated based on the master secret and “server finish” label.
8. The provider app generates an AuthChallenge METHOD_RETURN message and passes “DATA <s_rand:s_verfier>” as parameter to that message. The provider app sends the method reply to the consumer app via the AllJoyn router.
9. The consumer app invokes the AuthListener callback registered by the application to request a PIN. The AuthListener returns the PIN.
10. The consumer app computes a premaster secret based on the PIN.
11. The consumer app computes a master secret based on the premaster secret, c_rand, and s_rand as per the algorithm in section 8.1 of RFC 5246.
12. The consumer app generates the “server finish” verifier using the same algorithm as the provider app, and verifies that the computed value is same as the received s_verifier.
13. The consumer app computes a “client finish” c_verifier as per the algorithm in section 7.4.9 of RFC 5246. The c_verifier is generated based on the master secret and “client finish” label.
14. The consumer app generates an AuthChallenge METHOD_CALL message to send c_verifier to the server. The consumer app passes “DATA <c_verifier>”as parameter to the method call. The consumer app sends method call to the provider (server) app via the AllJoyn router.
15. The provider app generates the “client finish” verifier using the same algorithm as the consumer app, and verifies that the computed value is same as the received c_verifier.

    At this point, the client and server have authenticated with each other.
16. The provider app generates an AuthChallenge METHOD_RETURN message indicating that authentication is complete. The provider app passes “OK <s_GUID>” as parameter in that message, where s_GUID is the auth GUID of the provider app. The provider app sends the method reply to the consumer app via the AllJoyn router.
17. The consumer app sends an AuthChallenge METHOD_CALL to the provider app specifying “BEGIN <c_GUID>” as parameter. This indicates to the provider that the

client has received the OK message, and the stream of data messages is about to begin. The c_GUID is auth GUID of the consumer app.

18. The provider app sends an AuthChallenge METHOD_RETURN message specifying “BEGIN” as parameter.

8.6.6 ECDHE key exchanges

In the 14.06 release, new Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) based auth mechanism were added. For details on ECDHE-based auth mechanisms, refer to [4].

8.7 Generation of the session key

Figure 8-9 shows the message flow for the generation of session keys between peer applications.

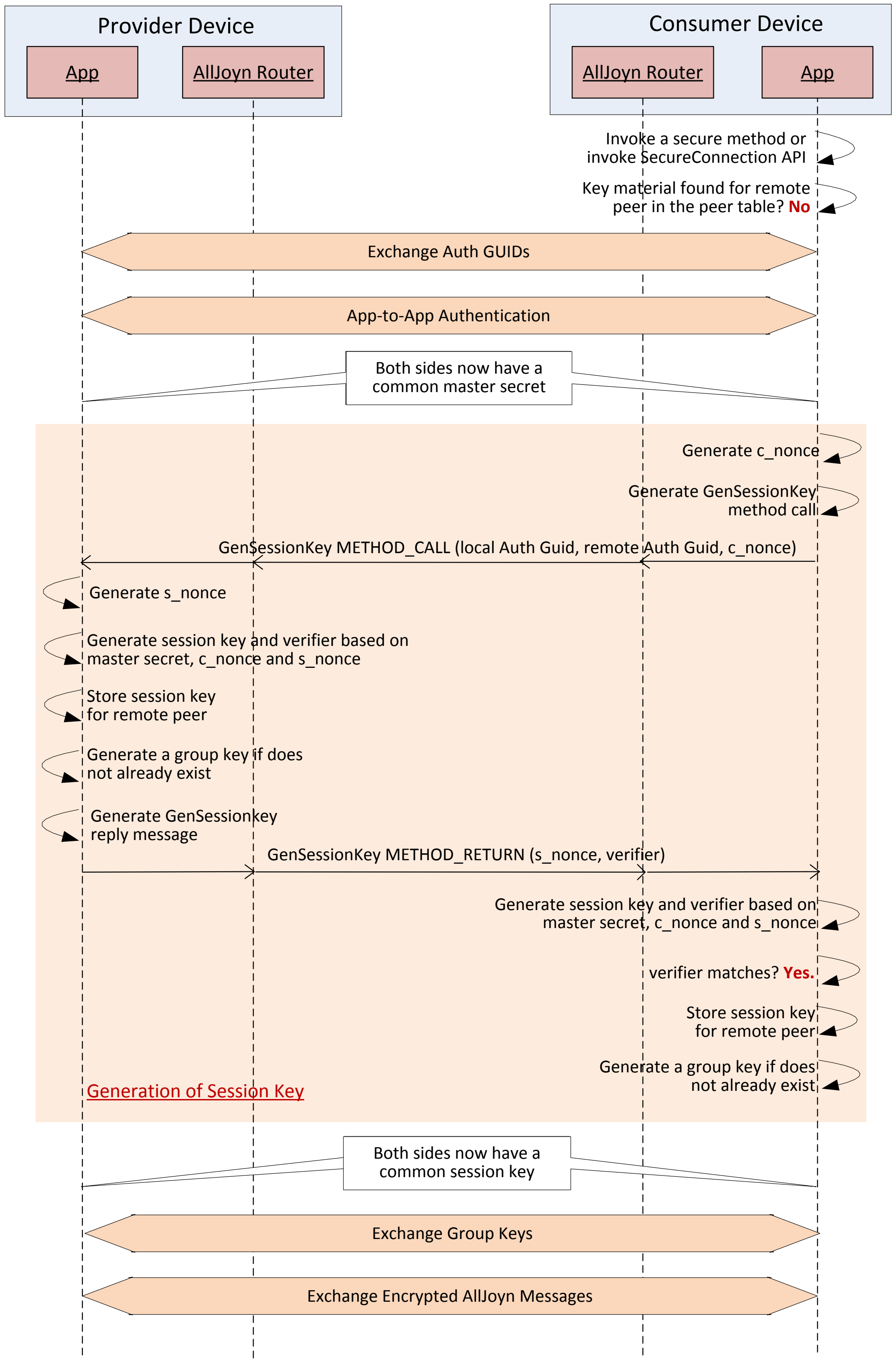

Provider Device
App
AllJoyn Router
Consumer Device
AllJoyn Router
App
Invoke a secure method or invoke SecureConnection API
Key material found for remote peer in the peer table? No
Exchange Auth GUIDs
App-to-App Authentication
Both sides now have a common master secret
Generate c_nonce
Generate GenSessionKey method call
GenSessionKey METHOD_CALL (local Auth Guid, remote Auth Guid, c_nonce)
Generate s_nonce
Generate session key and verifier based on master secret, c_nonce and s_nonce
Store session key for remote peer
Generate a group key if does not already exist
Generate GenSessionkey reply message
GenSessionKey METHOD_RETURN (s_nonce, verifier)
Generate session key and verifier based on master secret, c_nonce and s_nonce
verifier matches? Yes.
Store session key for remote peer
Generate a group key if does not already exist
Generation of Session Key
Both sides now have a common session key
Exchange Group Keys
Exchange Encrypted AllJoyn Messages

Figure 8-9. Session key generation between peer applications

The message flow steps are described below.

1. The consumer app generates a 28 bytes client nonce string c_nonce.
2. The consumer app generates a GenSessionKey METHOD_CALL message and sends it to the provider app via the AllJoyn router. This message includes local auth GUID corresponding to the consumer app, a remote auth GUID corresponding to the provider app, and c_nonce.
3. The provider app generates a 28 bytes server nonce string s_nonce.
4. The provider app generates a session key and a verifier based on the master secret, c_nonce, and s_nonce using the algorithm described in section 6.3 of RFC 5246. The “session key” label is used to generate the key.
5. The provider app stores the session key in the peer state table for the auth GUID associated with the consumer app.
6. The provider app generates a 128 bit group key if no group key exists for the provider app, and stores in the peer state table.
7. The provider app generates a GenSessionKey METHOD_RETURN message and sends it to the consumer app via the AllJoyn router. This message includes s_nonce and verifier.
8. The consumer app generates a session key and a verifier based on the master secret, c_nonce, and s_nonce using the same algorithm as the provider app (per section 6.3 of RFC 5246). The “session key” label is used for generate the key.
9. The consumer app verifies that the computed verifier is the same as the received verifier.
10. The consumer app stores the session key in the peer state table for the auth GUID associated with the provider app.
11. The consumer app generates a 128 bit group key if no group key exists for the consumer app, and stores in the peer state table.

The peer apps now have a common session key that can be used to exchange encrypted messages.

8.8 Exchange of group keys

Figure 8-10 shows the message flow for the exchange of group keys between peer applications. This is achieved via the ExchangeGroupKeys method call which is the first encrypted message sent between peer applications after the session key is established.

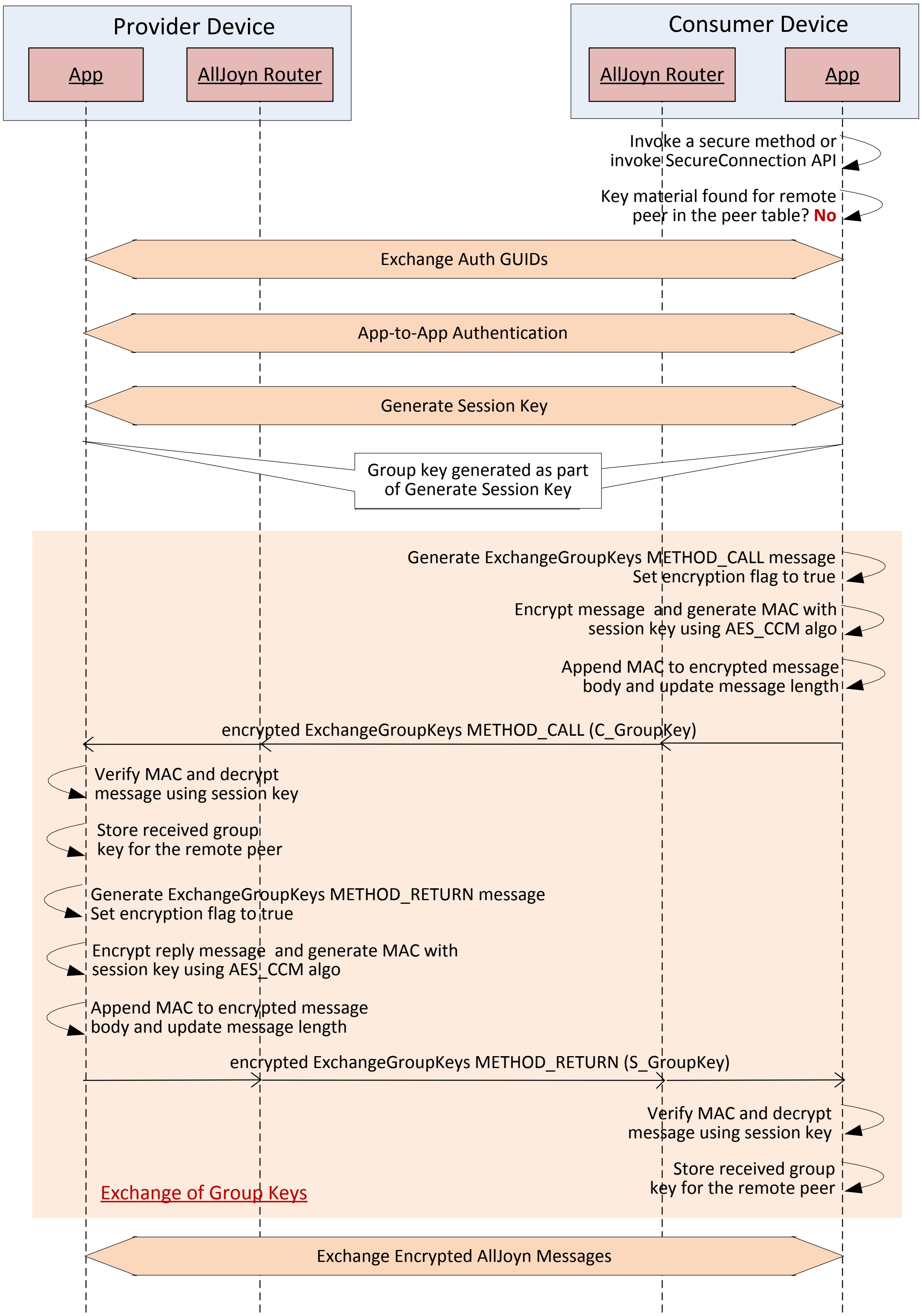

Provider Device
App
AllJoyn Router
Consumer Device
AllJoyn Router
App
Invoke a secure method or
invoke SecureConnection API
Key material found for remote
peer in the peer table? No
Exchange Auth GUIDs
App-to-App Authentication
Generate Session Key
Group key generated as part
of Generate Session Key
Generate ExchangeGroupKeys METHOD_CALL message
Set encryption flag to true
Encrypt message and generate MAC with
session key using AES_CCM algo
Append MAC to encrypted message
body and update message length
encrypted ExchangeGroupKeys METHOD_CALL (C_GroupKey)
Verify MAC and decrypt
message using session key
Store received group
key for the remote peer
Generate ExchangeGroupKeys METHOD_RETURN message
Set encryption flag to true
Encrypt reply message and generate MAC with
session key using AES_CCM algo
Append MAC to encrypted message
body and update message length
encrypted ExchangeGroupKeys METHOD_RETURN (S_GroupKey)
Verify MAC and decrypt
message using session key
Store received group
key for the remote peer
Exchange of Group Keys
Exchange Encrypted AllJoyn Messages

Figure 8-10. Exchange of group keys

The message flow steps are described below.

1. The consumer app generates an ExchangeGroupKeys METHOD_CALL message. This message includes the group key of the consumer app. The consumer app sets the encryption flag to true for this message.
2. The consumer app encrypts the message and generates an 8 bytes MAC (Message Authentication Code) using the session key for the remote peer app. Message encryption is done using AES CCM algorithm.
3. The consumer app appends the MAC to the encrypted message body and updates the message length to reflect the MAC.
4. The consumer app sends the encrypted ExchangeGroupKeys METHOD_CALL message to the provider app via the AllJoyn router.
5. The provider app verifies the MAC and decrypts the message using the session key stored for the consumer app.
6. The provider app stores the received group key for the remote peer (consumer app) in the peer state table.
7. The provider app generates an ExchangeGroupKeys METHOD_RETURN message. This message includes the group key of the provider app. The provider app sets the encryption flag to true for this message.
8. The provider app encrypts the message and generates an 8 bytes MAC using the session key for the remote peer app. Message encryption is done using AES CCM algorithm.
9. The provider app appends the MAC to the encrypted message body and updates the message length to reflect the MAC.
10. The provider app sends the encrypted ExchangeGroupKeys METHOD_RETURN message to the provider app via the AllJoyn router.
11. The consumer app verifies the MAC and decrypts the reply message using session key stored for the provider app.
12. The consumer app stores the received group key for the remote peer (provider app) in the peer state table.

Now the two apps have group key for each other which can be used to decrypt broadcast signal messages received from the peer application.

8.9 Exchange of encrypted messages

Once encryption credentials are established between applications, they can exchange encrypted methods and signals. These use cases are captured below.

8.9.1 Encrypted method call

Figure 8-11 shows the message flow for exchange of encrypted method call/reply between the consumer and provider applications. The reply message to an encrypted method call is also sent encrypted.

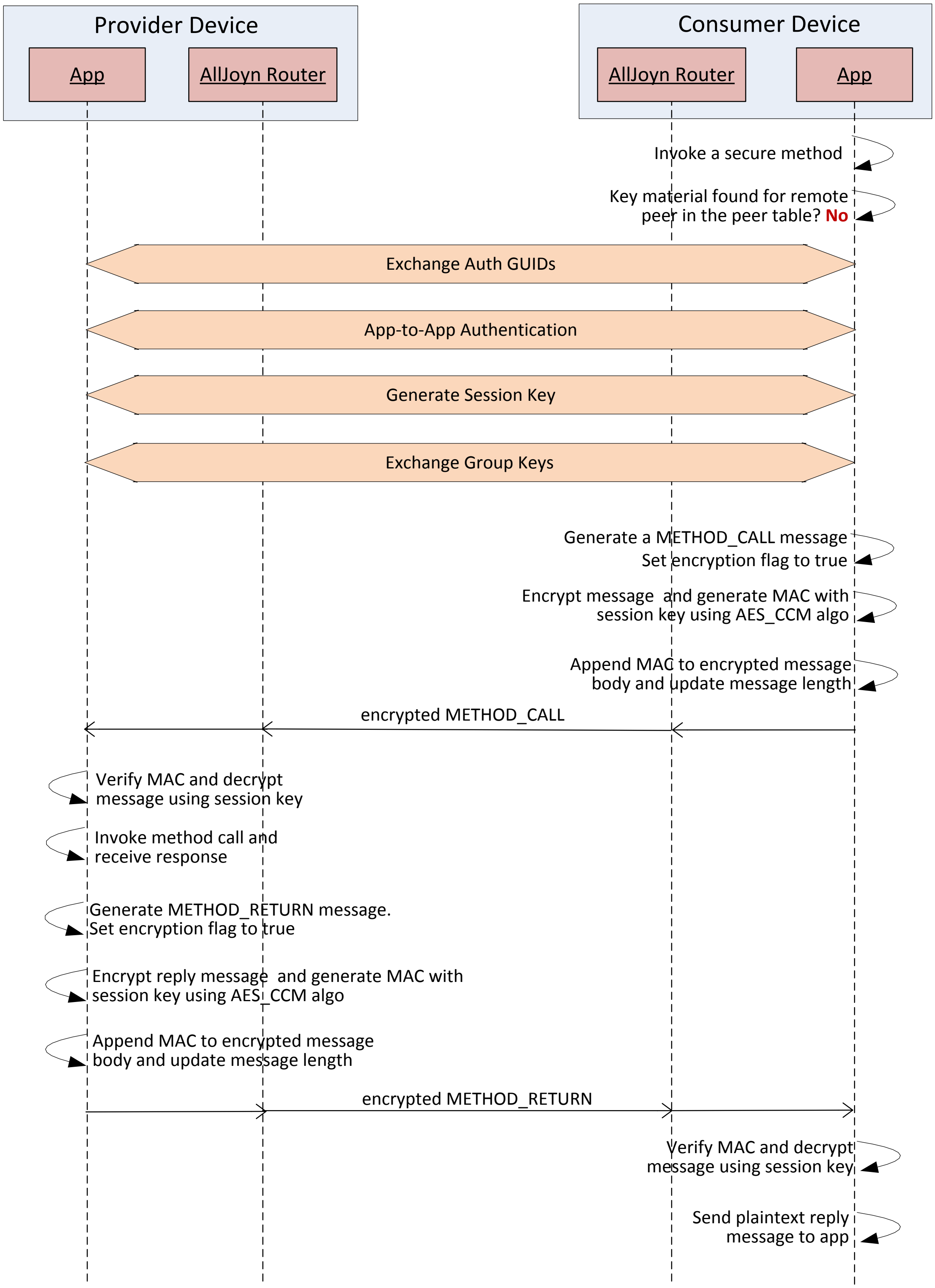

Provider Device
App
AllJoyn Router
Consumer Device
AllJoyn Router
App
Invoke a secure method
Key material found for remote peer in the peer table? No
Exchange Auth GUIDs
App-to-App Authentication
Generate Session Key
Exchange Group Keys
Generate a METHOD_CALL message Set encryption flag to true
Encrypt message and generate MAC with session key using AES_CCM algo
Append MAC to encrypted message body and update message length
encrypted METHOD_CALL
Verify MAC and decrypt message using session key
Invoke method call and receive response
Generate METHOD_RETURN message. Set encryption flag to true
Encrypt reply message and generate MAC with session key using AES_CCM algo
Append MAC to encrypted message body and update message length
encrypted METHOD_RETURN
Verify MAC and decrypt message using session key
Send plaintext reply message to app

Figure 8-11. Encrypted method call/reply

The message flow steps are described below.

1. The consumer app generates a METHOD_CALL message for the secure method and sets the encryption flag to true for this message.
2. The consumer app encrypts the message and generates an 8 bytes MAC using the session key for the destination app. Message encryption is done using AES CCM algorithm.
3. The consumer app appends the MAC to the encrypted message body and updates the message length to reflect the MAC.
4. The consumer app sends the encrypted METHOD_CALL message to the provider app via the AllJoyn router.
5. The provider app verifies the MAC and decrypts the message using session key stored for the consumer app.
6. The provider app's AllJoyn core library invokes the MethodCall handler, which invokes the method call on the service object interface and receives a reply.
7. The provider app generates a METHOD_RETURN message for the reply and sets the encryption flag to true for this message.
8. The provider app encrypts the message and generates an 8 bytes MAC using the session key for the consumer app. Message encryption is done using AES CCM algorithm.
9. The provider app appends the MAC to the encrypted message body and updates the message length to reflect the MAC.
10. The provider app sends the encrypted METHOD_RETURN message to the consumer app via the AllJoyn router.
11. The consumer app verifies the MAC and decrypts the reply message using session key stored for the provider app.
12. The consumer app's AllJoyn core library sends the plaintext reply message to the application.

8.9.2 Encrypted signal

Figure 8-12 shows the message flow for sending an encrypted session based signal from provider application to consumer applications. The signal can be sent to a destination (unicast signal) or to multiple endpoints as multicast/broadcast signals.

NOTE

Sessionless signals are not sent encrypted in current AllJoyn system. In future, implementation can be enhanced to encrypt sessionless signals as well.

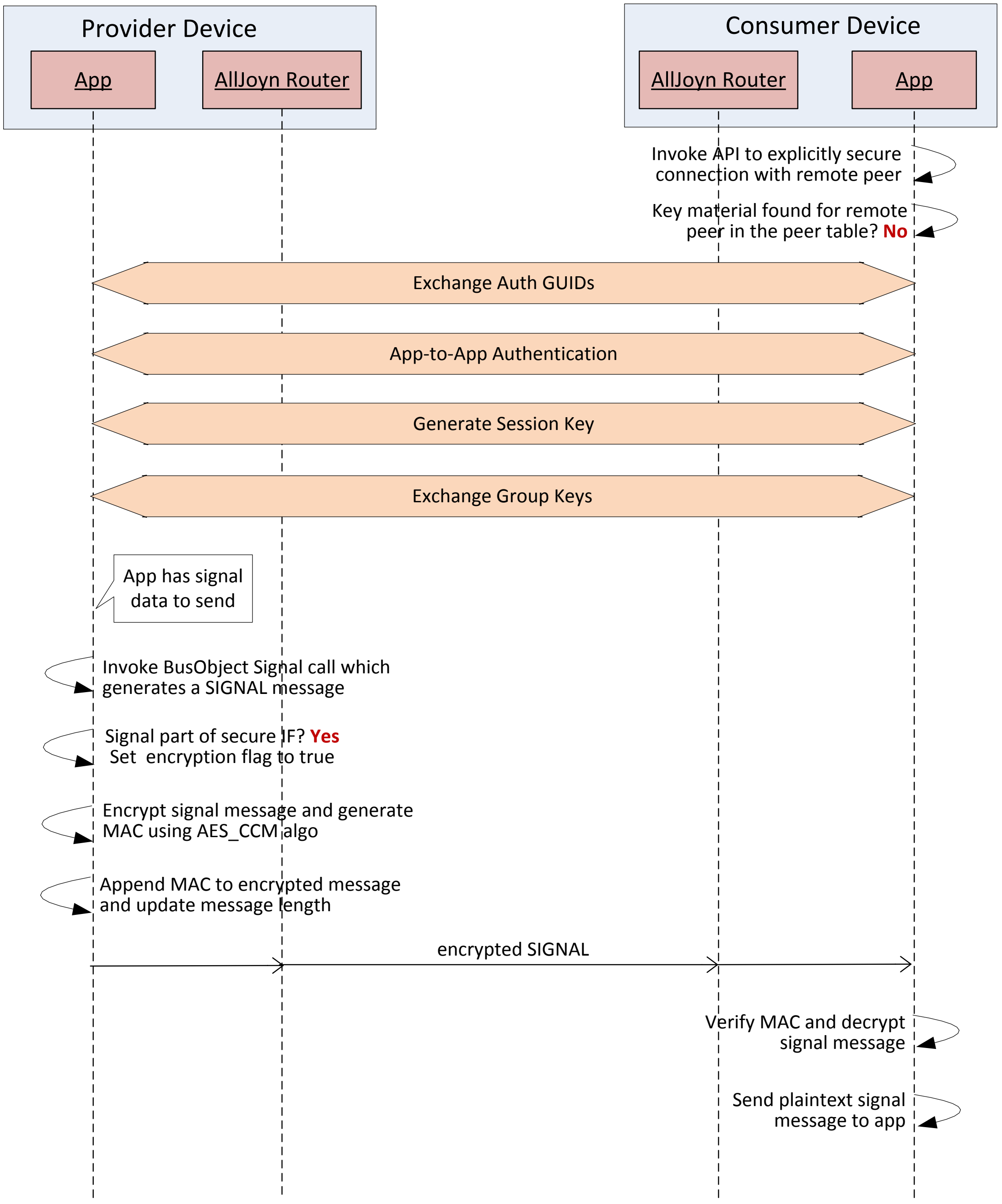

Provider Device
App
AllJoyn Router
Consumer Device
AllJoyn Router
App
Invoke API to explicitly secure connection with remote peer
Key material found for remote peer in the peer table? No
Exchange Auth GUIDs
App-to-App Authentication
Generate Session Key
Exchange Group Keys
App has signal data to send
Invoke BusObject Signal call which generates a SIGNAL message
Signal part of secure IF? Yes
Set encryption flag to true
Encrypt signal message and generate MAC using AES_CCM algo
Append MAC to encrypted message and update message length
encrypted SIGNAL
Verify MAC and decrypt signal message
Send plaintext signal message to app

Figure 8-12. Encrypted signal

The message flow steps are described below.

1. The consumer and provider apps have already authenticated and established encryption keys with each other.
2. The provider app has some signal data to send. It invokes the BusObject Signal() call which generates a SIGNAL message.

3. The provider app sets the encryption flag to true for the SIGNAL message if the signal is defined as part of a secure interface.
4. The provider app encrypts the SIGNAL message and generates an 8 bytes MAC using either the group key or session key as per the logic in Figure 8-13. Message encryption is done using AES CCM algorithm.
5. The provider app appends the MAC to the encrypted SIGNAL message body and updates the message length to reflect the MAC.
6. The provider app sends the encrypted SIGNAL message to the consumer app via the AllJoyn router.
7. The consumer app verifies the MAC and decrypts the SIGNAL message using either the session key or group key as per the logic in Figure 8-14.
8. The consumer app's AllJoyn core library sends the plaintext signal message to the application.

8.9.2.1 Key selection logic

On the provider application side, unicast signals get encrypted using the session key and multicast/broadcast signals get encrypted using group key. Figure 8-13 shows the key selection logic for encrypting signals.

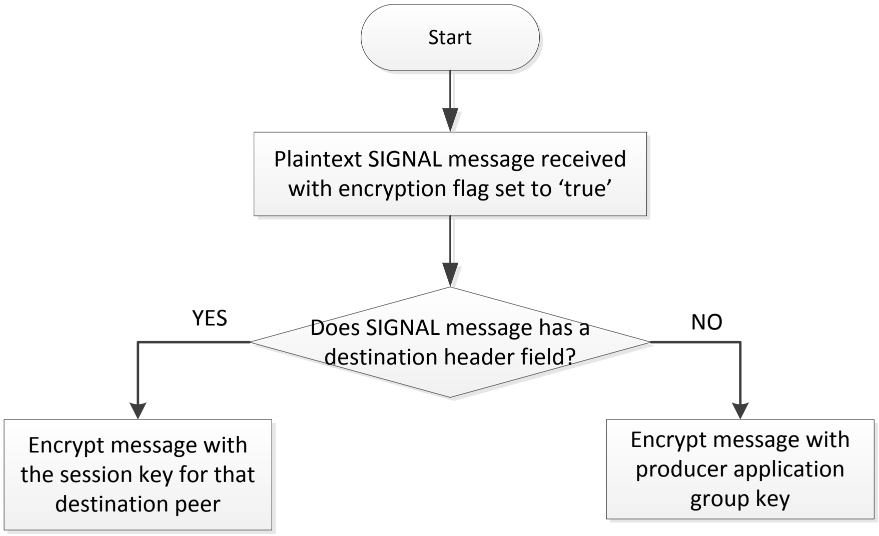

Start
Plaintext SIGNAL message received with encryption flag set to 'true'
Does SIGNAL message has a destination header field?
YES
NO
Encrypt message with the session key for that destination peer
Encrypt message with producer application group key

Figure 8-13. Key selection for signal encryption (on the provider app)

On the consumer side, a reverse logic is applied for selecting key for decrypting received signals messages as shown in Figure 8-14.

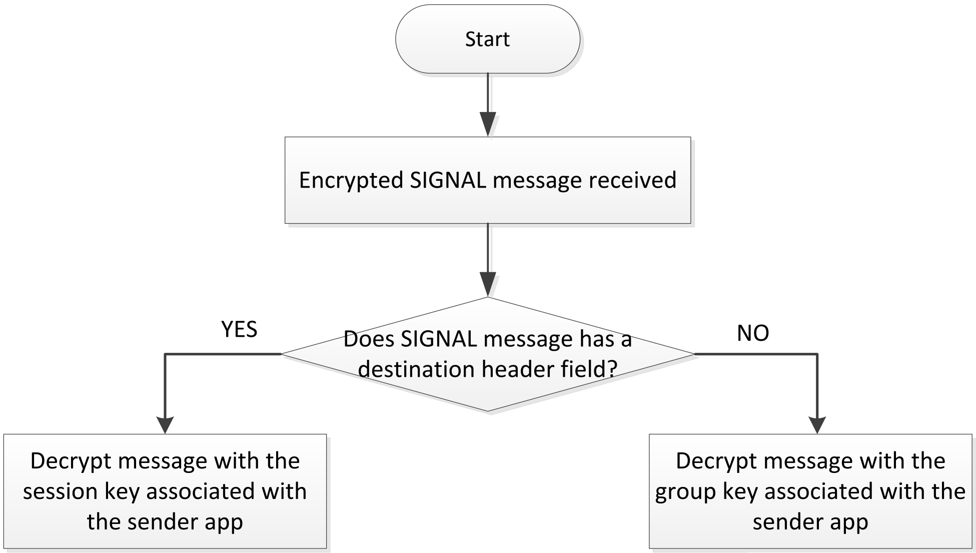

Start
Encrypted SIGNAL message received
Does SIGNAL message has a destination header field?
YES
NO
Decrypt message with the session key associated with the sender app
Decrypt message with the group key associated with the sender app

Figure 8-14. Key selection for signal decryption (on the consumer app)

8.10 org.alljoyn.Bus.Peer.Authentication interface

The org.alljoyn.Bus.Peer.Authentication interface is the AllJoyn interface between two AllJoyn core libraries that support the application layer security within AllJoyn.

Table 8-4 summarizes members from org.alljoyn.Bus.Peer.Authentication interface.

Table 8-4. org.alljoyn.Bus.Peer.Authentication interface methods

Method	Parameters			Description
	Parameter name	Direction	Description	
ExchangeGuids	localGuid	in	Auth GUID for the initiator application	Method for an application to exchange its auth GUID and authentication protocol version with a remote peer application.
	localVersion	in	Auth version for the initiator application	
	remoteGuid	out	Auth GUID for the remote peer application	
	remoteVersion	out	Auth version for the remote peer application	

Method	Parameters			Description
	Parameter name	Direction	Description	
AuthChallenge	challenge	in	Auth data provided by the initiator app	Method for an application to initiate authentication and exchange authentication data with a remote peer application.
	response	out	Auth data returned by the provider app	
GenSessionKey	localGuid	in	Auth GUID for the initiator application	Method for an application to generate a session key with a remote peer application.
	remoteGuid	out	Auth GUID for the remote peer application	
	localNonce	in	Nonce generated by the initiator app	
	remoteNonce	out	Nonce generated by the remote peer app	
	verifier	out	Verifier generated by the remote peer app	
ExchangeGroupKeys	localKeyMatter	in	Group key of the initiator app	Method for an application to exchange group key with a remote peer application.
	remoteKeyMatter	out	Group key of the remote peer app	

9 Thin Apps

9.1 Overview

The AllJoyn system is designed to operate across AllJoyn-enabled devices with different capabilities. The AllJoyn Standard Core Library (AJSCL) is designed to run on devices that usually have significant amounts of memory, available energy, and computing power, along with operating systems that support multiple processes/threads with multiple standard language environments. The AJSCL is designed for general purpose computer devices and supports application running on HLOS including Microsoft Windows, Linux, Android, iOS, and OpenWRT.

On the other hand, single-purpose AllJoyn-enabled devices usually have an embedded system running on a microcontroller designed to provide specific functionality. Such embedded systems are optimized to reduce the size and cost of the product, often by limiting memory size, processor speed, available power, peripherals, user interfaces, or all of the above. The AllJoyn Thin Core Library (AJTCL) is designed to bring the benefits of the AllJoyn distributed programming environment to embedded system-based devices.

The AJTCL provides a lightweight implementation of core AllJoyn functionality for embedded microcontroller applications. An embedded system-based AllJoyn device (thin AllJoyn device) only includes an AllJoyn thin application utilizing the AJTCL and does not include an AllJoyn router component because of its resource-constrained environment. It borrows an AllJoyn router from another standard AllJoyn-enabled device in the AllJoyn proximal network, and uses it for core AllJoyn functions including advertisement and message routing. An AllJoyn thin application is fully compatible and interoperable with standard AllJoyn applications on the AllJoyn proximal network. In fact, a remote application will not even know that it is talking with an AllJoyn thin application on the other side.

Figure 9-1 shows a context architecture depicting how AllJoyn thin applications fit in the overall AllJoyn distributed system. It shows two thin AllJoyn-enabled devices (device 3 and device 4) with a single AllJoyn thin application installed on each of them. A thin app is built on top of AJTCL and it connects with the distributed AllJoyn bus by establishing a connection with an AllJoyn router on a standard AllJoyn-enabled device (e.g., AllJoyn router installed on the Wi-Fi Access Point). The AJTCL uses the AllJoyn service advertisement and discovery process to discover the AllJoyn router via a BusNode well-known name. After the discovery phase, the AJTCL establishes a connection with the discovered AllJoyn router over TCP. Once connected with the AllJoyn router, the thin app is just like any other application endpoint on the AllJoyn distributed bus.

NOTE

More than one thin application can connect to a given AllJoyn router.

A thin app can act as an AllJoyn service provider, an AllJoyn service consumer or both. It follows the same session establishment procedures as AllJoyn standard apps to accept sessions from and/or connect to sessions with other remote apps, which can be another AllJoyn thin app or AllJoyn standard app.

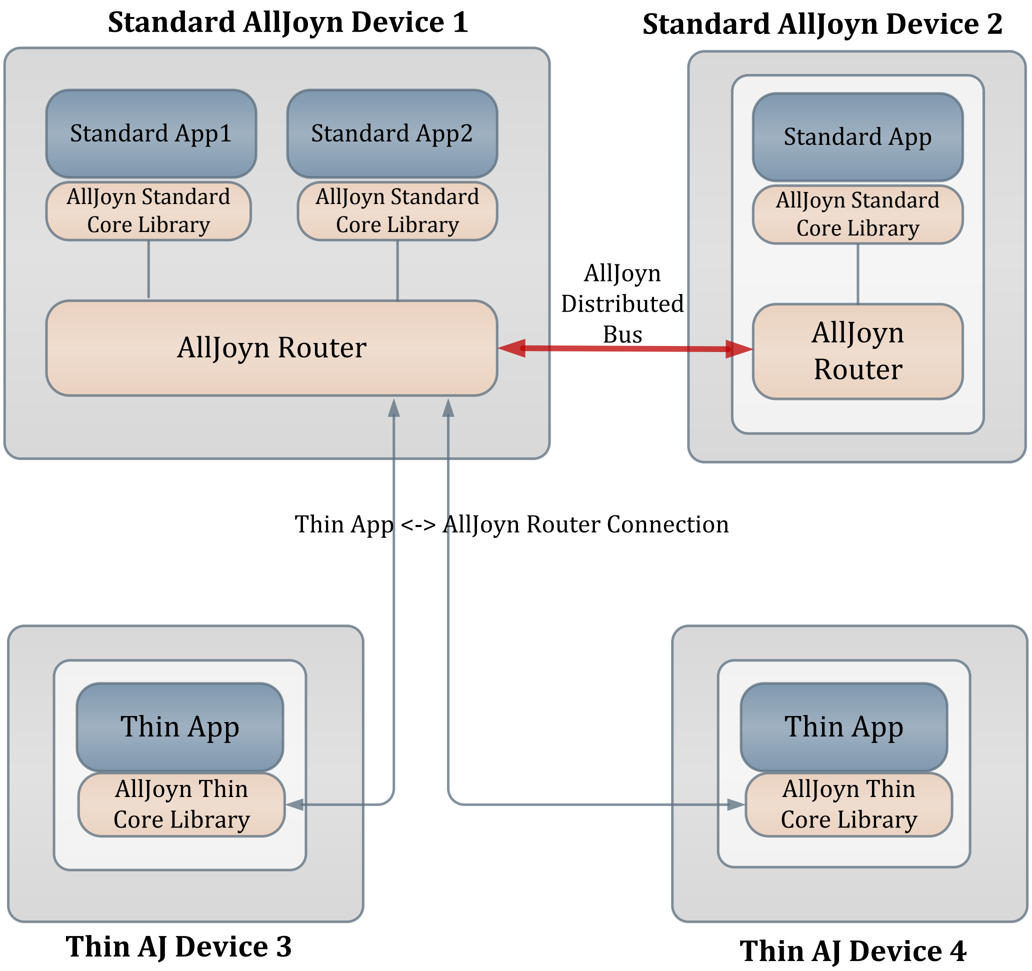

Standard AllJoyn Device 1
Standard App1
Standard App2
AllJoyn Standard Core Library
AllJoyn Standard Core Library
AllJoyn Router
Standard AllJoyn Device 2
Standard App
AllJoyn Standard Core Library
AllJoyn Router
AllJoyn Distributed Bus
Thin App <-> AllJoyn Router Connection
Thin App
AllJoyn Thin Core Library
Thin AJ Device 3
Thin App
AllJoyn Thin Core Library
Thin AJ Device 4

Figure 9-1. Thin app context architecture

9.2 Functional architecture

Figure 9-2 shows the detailed functional architecture for an AllJoyn thin application. A thin app includes app-specific code (app code) and the AJTCL. As part of the app code, a thin app can include one or more AllJoyn service frameworks which include Onboarding, Configuration, and Notification service frameworks. App Code also includes app-specific AllJoyn services if the thin app is acting as an AllJoyn service provider.

The AJTCL consists of some key functional modules as shown in Figure 9-2, among other supported functions. These include Bus Connection Manager, About, Messaging and App Authentication modules.

- The Bus Connection Manger module provides discovery and connection establishment with a nearby AllJoyn router (BusNode).
- The About module provides advertisement and discovery functions for thin app. It supports sending out the Announcement sessionless signal for the thin app over distributed AllJoyn bus.

- The Messaging module provides marshaling/unmarshaling for AllJoyn messages and routing these to the connected AllJoyn router.
- The App Authentication module provides application-level authentication and security between thin app and remote AllJoyn apps. The ALLJOYN_PIN_KEYX auth mechanism is supported in the AJTCL for releases before the 14.06 release. This auth mechanism is removed from AJTCL in the 14.06 release. Starting from the 14.06 release, the AJTCL supports a new set of Elliptic Curve Diffie-Hellman Ephemeral (ECDHE)-based auth mechanisms as described in section 9.5.

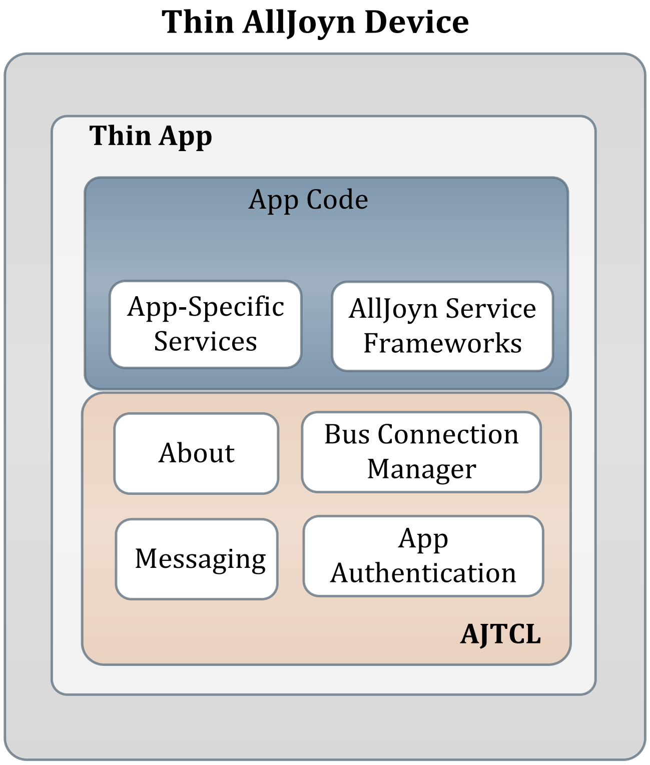


Figure 9-2. Thin App functional architecture

9.3 AJTCL to AllJoyn router connection

Upon startup, the thin application initiates the process of discovery and connection establishment with an AllJoyn router on another standard AllJoyn-enabled device. This is done using the name-based discovery mechanism.

An AllJoyn router that supports hosting connections for thin apps advertises a BusNode well-known name. The advertised well-known name can be one or both of the following:

- Generic BusNode well-known name “org.alljoyn.BusNode” driven by the AllJoyn router configuration
- Specific BusNode well-known name advertised by an application attached to the AllJoyn router, meant for discovery only by related thin applications.

The AllJoyn router advertises the BusNode well-known name quietly, which means that these advertisement messages are not sent out gratuitously by the AllJoyn router. The AllJoyn router only sends out the BusNode well-known name advertisement in response to a query from a thin app. Also, the advertisement message is sent out quietly via unicast back to the requester (instead of being sent over multicast). This logic is meant to minimize the network traffic generated as a result of thin app related discovery.

The AllJoyn router supports only untrusted relationships with the AJTCL, in which it allows a connection from any AJTCL via SASL anonymous authentication. No real credential is exchanged between the AJTCL and the AllJoyn router. This is fine because app level authentication is supported by thin app.

The AllJoyn router limits the number of simultaneous connections with thin applications in the AllJoyn network. This limit is configurable as ‘number of untrusted connections’ via the router config file. The AllJoyn router stops advertising all BusNode names when the ‘number of untrusted connections’ limit is reached. It starts advertising BusNode names again when the current number of thin app connections goes below the ‘number of untrusted connections’ limit.

Figure 9-3 shows the message flow for the AJTCL discovering and connecting with the AllJoyn router. The connection process is split into the following phases:

- Discovery phase: The AJTCL discovers an AllJoyn router on the AllJoyn proximal network via the BusNode name-based discovery mechanism. The AJTCL sends out a WHO-HAS message for the BusNode well-known name following a backoff schedule. The IS-AT message is sent over unicast to the AJTCL by the AllJoyn router advertising that BusNode name.
- Connection phase: The AJTCL establishes a TCP connection with the AllJoyn router based on the connection details received in the IS-AT message.
- Authentication phase: SASL anonymous authentication is used by the AJTCL to authenticate and start using services of the AllJoyn router.

The AJTCL also exchanges the AllJoyn protocol version with the AllJoyn router. If the AllJoyn router supports an older AllJoyn protocol version, the connection process fails. For the first-time connecting with any AllJoyn router, this process also generates a local GUID for the AJTCL and sends to the AllJoyn router.

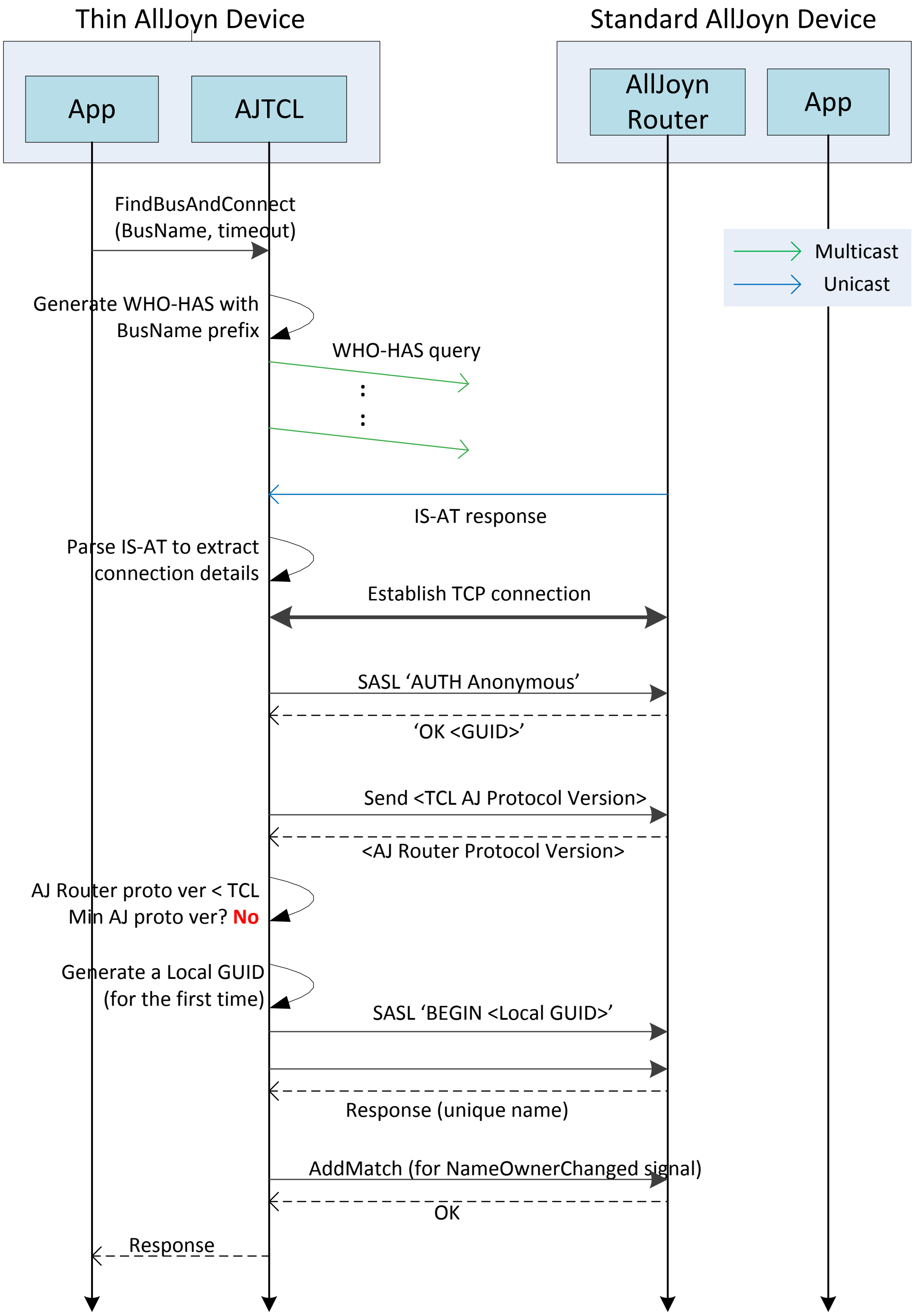

Thin AllJoyn Device
App
AJTCL
Standard AllJoyn Device
AllJoyn Router
App
Multicast
Unicast
FindBusAndConnect (BusName, timeout)
Generate WHO-HAS with BusName prefix
WHO-HAS query
:
:
IS-AT response
Parse IS-AT to extract connection details
Establish TCP connection
SASL 'AUTH Anonymous'
'OK <GUID>'
Send <TCL AJ Protocol Version>
<AJ Router Protocol Version>
AJ Router proto ver < TCL Min AJ proto ver? No
Generate a Local GUID (for the first time)
SASL 'BEGIN <Local GUID>'
Response (unique name)
AddMatch (for NameOwnerChanged signal)
OK
Response

Figure 9-3. Thin app discovering and connecting to the AllJoyn router

9.3.1 AJTCL and AllJoyn router compatibility

Table 9-1 captures the compatibility matrix between the AJTCL and AllJoyn router across the AllJoyn 14.02 and 14.06 releases. The AJTCL using the 14.06 release is only compatible with a 14.02 AllJoyn router if the router does not require AJTCL authentication. The thin app also has the min AJ Protocol version set to 8 (same as the 14.02 AllJoyn router), implying that it is not using the NGNS feature.

Table 9-1. AJTCL and AllJoyn router compatibility

Router **AJTCL**	**14.02** **(AJTCL auth enabled)**	**14.02** **(AJTCL auth disabled)**	**14.06**
14.02	Compatible	Compatible	Compatible
14.06 **(thin app not using NGNS)**	Not-compatible	Compatible	Compatible
14.06 **(thin app using NGNS)**	Not-compatible	Not-compatible	Compatible

9.3.2 WHO-HAS message schedule

The AJTCL supports a retry schedule for sending WHO-HAS discovery messages. The retry schedule follows a backoff schedule as follows:

1. Send the WHO-HAS message once a second for 10 seconds.
2. Wait 10 seconds, then send another WHO-HAS message.
3. Wait 20 seconds, then send another WHO-HAS message.
4. Wait 40 seconds, then send another; repeat until the overall discovery timeout expires.

NOTE

The overall discovery timeout is specified by the thin app in the FindBusAndConnect() API call.

9.3.3 Detecting link failures

The AJTCL provides a mechanism for the thin application to implement a probing mechanism to detect connectivity failures with the AllJoyn router. This can be achieved by invoking the SetBusLinkTimeout() API provided by the AJTCL. The thin app specifies a timeout value (with minimum timeout of 40 seconds) as part of this API. If no link activity is detected during this time period, the AJTCL sends probe packets every 5 seconds over the router link. If no acknowledgment is received for three consecutive probe packets, an error is returned to the thin application.

At this point, the thin app should reinitiate discovery for the AllJoyn router.

9.4 Thin app functionality

As mentioned previously, the AJTCL supports all of the key AllJoyn core functionality as a standard core library. APIs are provided as part of the AJTCL for the thin app to invoke core functionality. The AJTCL in turn generates appropriate AllJoyn format messages (for method_call/reply, signals etc.) to invoke related APIs on the AllJoyn router. The AJTCL sends the generated AllJoyn messages to the AllJoyn router to accomplish the given functionality. The thin app message flow for core functionality is similar to the standard app with the key difference that the thin app is connected remotely with the AllJoyn router.

Figure 9-4 shows an example message flow for a thin app discovering a well-known name prefix.

NOTE

The AJTCL and AllJoyn router exchange data using AllJoyn messages (method_call/reply and signals).

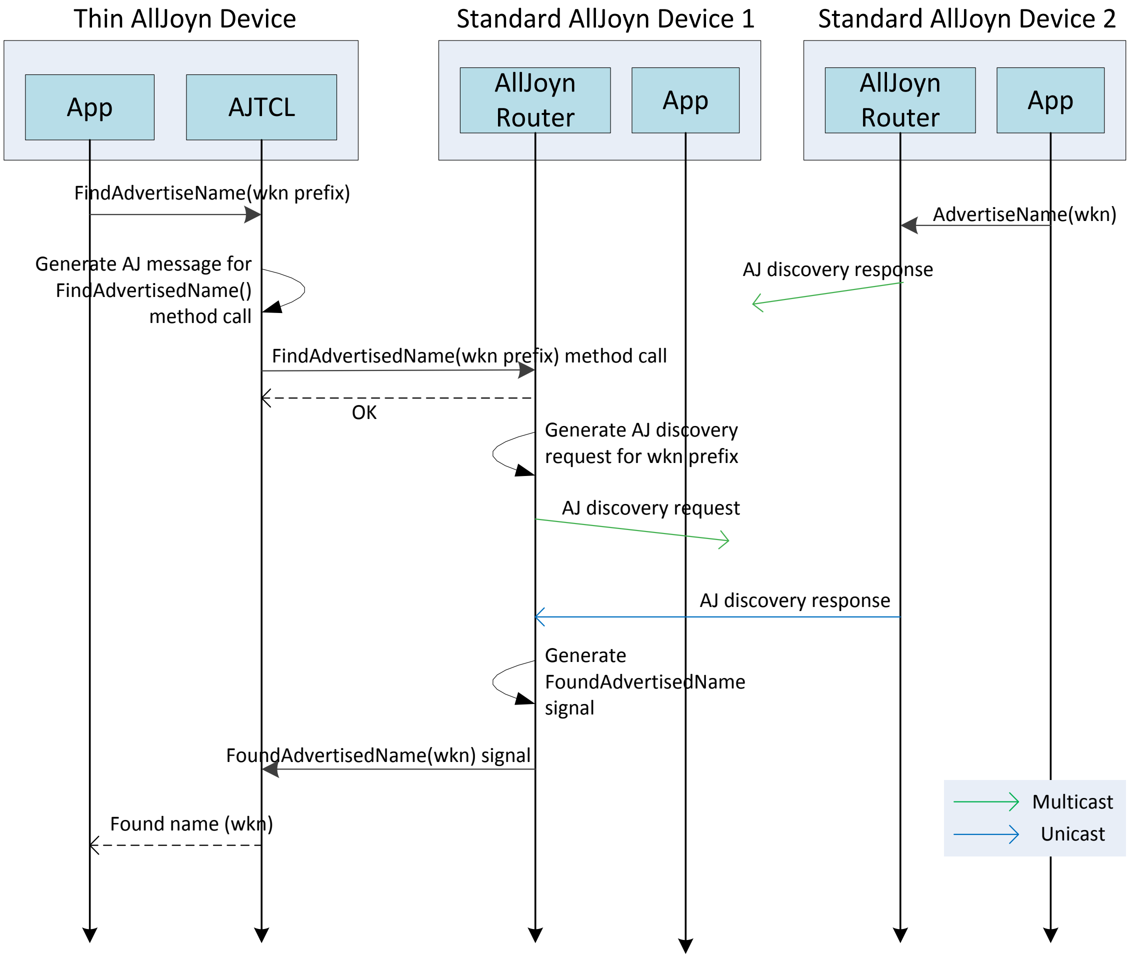

Thin AllJoyn Device
Standard AllJoyn Device 1
Standard AllJoyn Device 2
App
AJTCL
AllJoyn Router
App
AllJoyn Router
App
FindAdvertiseName(wkn prefix)
AdvertiseName(wkn)
Generate AJ message for FindAdvertisedName() method call
AJ discovery response
FindAdvertisedName(wkn prefix) method call
OK
Generate AJ discovery request for wkn prefix
AJ discovery request
AJ discovery response
Generate FoundAdvertisedName signal
FoundAdvertisedName(wkn) signal
Found name (wkn)
Multicast
Unicast

Figure 9-4. Thin app discovering a well-known name prefix

The AJTCL provides support for following core AJ functionality:

- Service Discovery and Advertisement: Both legacy Name Service and Next-Gen Name Service functions are supported.
- About advertisement

- Session establishment
- Sessionless signals
- App layer authentication
  - The AJTCL provides app layer authentication so that thin app can implement secure interfaces and also access secure interfaces on other AllJoyn providers.
  - New authentication schemes are supported in the 14.06 release (see section 9.5).

Thin apps can also include existing AllJoyn service frameworks functionality by bundling thin app-specific libraries provided for these service frameworks.

9.5 App layer authentication

The AJTCL provides support for app layer authentication for the thin app to implement and access secure AllJoyn services. App layer authentication schemes supported are different in release prior to the 14.06 release and starting from the 14.06 release as described below.

Prior to the 14.06 release, the AJTCL supports ALLJOYN _PIN_KEYX auth mechanism for app layer authentication. Also, SASL protocol is used for authentication.

Starting from the 14.06 release, ALLJOYN _PIN_KEYX auth mechanism is removed from AJTCL. New Elliptic Curve Diffie-Hellman Ephemeral (ECDHE)-based auth mechanism were added to the AJTCL:

- ECDHE_NULL is an anonymous key agreement. There is no PIN or passphrase required.
- ECDHE_PSK is a key agreement authenticated with a preshared key like a PIN, passphrase, or symmetric key.
- ECDHE_ECDSA is a key agreement authenticated with an asymmetric key validated with an ECDSA signature.

The use of SASL protocol for authentication is removed from the AJTCL in the 14.06 release. Instead, an AllJoyn-based protocol is used for app layer authentication.

9.5.1 Auth compatibility

A 14.06 thin app cannot interact with a 14.02 thin app over secure interfaces and vice versa because these apps support different types of auth mechanisms. These apps can still talk to each other over nonsecure interfaces.

Table 9-2 shows the thin app compatibility matrix across the 14.02 and 14.06 releases.

Table 9-2 Thin app compatibility

14.02 Provider **Thin App**	**14.06 Consumer** **Thin App**
With secure interfaces	Incompatible
With nonsecure Interfaces	Compatible

14.02 Provider **Thin App**	**14.06 Consumer** **Thin App**
14.06 Provider **Thin App**	**14.02 Consumer** **Thin App**
With secure interfaces	Incompatible
With nonsecure Interfaces	Compatible

10 Events and Actions

10.1 Overview

The Events and Actions feature is part of the AllJoyn core, designed to enable creating If-This-Then-That (IFTTT)-based rules logic in the AllJoyn network.

- Events are used by AllJoyn devices/apps to notify other AllJoyn devices/apps when something of significance occurs in the network.
- Actions enable specific responses to AllJoyn events detected in the AllJoyn network. In this regard, Events and actions go hand-in-hand. An action is a way for making an application or device do something.

For example, an AllJoyn application can broadcast an event signifying that something has happened, such as movement that was detected by a motion detector. An AllJoyn application can receive this event and respond to it by taking a specific action, such as turning on the security camera.

Events are realized using AllJoyn sessionless signals, while actions are realized using AllJoyn methods. A description element is added to the AllJoyn introspect XML format to provide human readable text for the Events and Actions feature.

Figure 10-1 illustrates the context architecture for the Events and Actions feature. Events and actions are advertised in the Announcement signal using the org.allseen.Inrospectable interface. Any advertised object supporting an event-emitting interface or action-receiving interface will include this new interface in the Announcement signal. The Authoring app is an AllJoyn-enabled app that provides a UI for creating IFTTT based rules for automation in the IoE network The Authoring app receives announcement signals from AllJoyn devices that emit events and/or can receive actions. The app introspects those devices to retrieve a human-readable description for events and actions as part of the enhanced introspection XML data.

These human readable text description details can be presented to a user allowing the user to create IFTTT based rules for automation in the IoE network. These IFTTT rules get configured on a Rule Engine which could be on the same device or a different device than the Authoring app.

NOTE

The Rules Engine is beyond the scope of current design and its implementation is left to the ecosystem). The Rules Engine app detects when the event is emitted. Based on the configured IFTTT rules, it executes actions (method call) on the action-receiving devices.

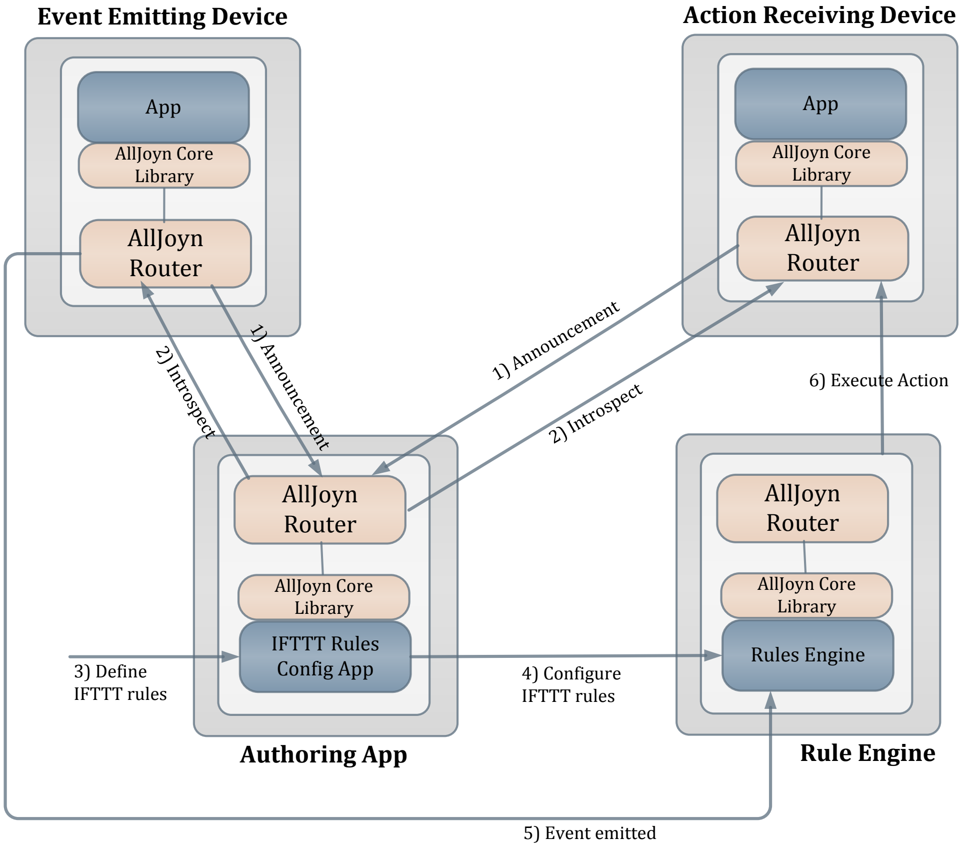


Figure 10-1. Events and Actions context architecture

10.2 Enhanced introspection XML

The AllJoyn system supports introspection XML format as defined by the D-Bus specification via org.freedesktop.DBus.Introspectable interface. To make events and actions discoverable, an enhanced AllJoyn introspection XML is made available to provide human readable description elements. This XML provides description elements as applicable under Objects, Interfaces, Methods (including parameters), Signals (including parameters) and Properties.

The description element may appear inside the enhanced introspection XML under the elements captured in Table 10-1.

Table 10-1. Elements carrying description element

XML element	Definition
Node	Objects and sub-objects within the tree of objects.
interface	Interface element
method	Method element
property	Property element
Signal	Signal element

XML element	Definition
arg	Arguments to signals and methods.

In addition, the enhanced Introspection XML also includes sessionless=”true|false” attribute for the signal element to indicate whether or not it is sessionless signal.

Figure 10-2 provides an example of enhanced introspection XML format.

```
<node name="/com/example/LightBulb">
         <description>Your lightbulb</description>
         <interface name="com.example.LightBulb">
             <description>Provides basic lighting functionality</description>
             <method name="ToggleSwitch">
               <description>Invoke this to toggle whether the light is on or
               off</description>
               <arg name="brightess" type="i" direction="in">
                 <description>A value to specify how bright the bulb should
                 shine</description>
               </arg>
             </method>
             <signal name="LightOn" sessionless=”true”>
                 <description>Emitted when the light turns on</description>
             </signal>
             <signal name="LightOff" sessionless=”true”>
                 <description>Emitted when the light turns off</description>
             </signal>
             <property name="LightState" type="y" access="read">
                 <description>The current state of this light bulb</description>
             </property>
         </interface>
         <node name="child">
           <description>Some relevant description</description>
         </node>
</node>
```

Figure 10-2. Enhanced Introspection XML example

10.3 Introspectable interface

The org.allseen.introspectable interface is designed to provide access to the enhanced introspection XML containing the description elements. Table 10-2 provides the definition for the org.allseen.introspectable interface which is implemented by all AllJoyn objects.

Table 10-2. Introspectable interface

Method name	Parameters			Description
	Parameter name	**Direction**	**Description**	
AttachSession	languageTags	out	A list of the languages in which this object has descriptions	Return the aggregate of the languages for which this object has descriptions. For example, if an object implements two interfaces, X and Y - X has all of its members described in English (en) and French (fr) and Y has some descriptions in English (en) and Chinese (cn), this method will return [“en”, “fr”, “cn”]. The language tags will comply with IETF language tag standards.
IntrospectWithDescription	languageTag	in	Requested Language	Returns the XML defined above with descriptions in the specified language (exact match only – no best match). If an element, e.g., method, does not have a description in that language, no description attribute is placed within the element.
	data	out	Returned introspection XML	