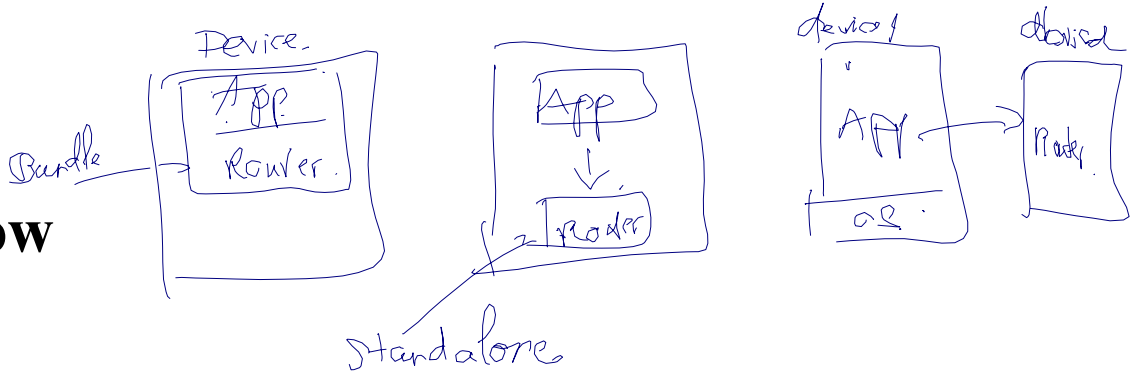


Must Know

LEARN

AllJoyn® is a collaborative open-source software framework that makes it easy for developers to write applications that can discover nearby devices, and communicate with each other directly regardless of brands, categories, transports, and OSes without the need of the cloud. The AllJoyn framework is extremely flexible with many features to help make the vision of the Internet of Things come to life.

Proximal Network

The AllJoyn framework handles the complexities of discovering nearby devices, creating sessions between devices, and communicating securely between those devices. It abstracts out the details of the physical transports and provides a simple-to-use API. Multiple connection session topologies are supported, including point-to-point and group sessions. The security framework is flexible, supporting many mechanisms and trust models. And the types of data transferred are also flexible, supporting raw sockets or abstracted objects with well-defined interfaces, methods, properties, and signals.

Flexible

One of the defining traits of the AllJoyn framework is its inherent flexibility.
It was designed to run on multiple platforms, ranging from small embedded RTOS platforms to full-featured OSes. It supports multiple language bindings and transports. And since the AllJoyn framework is open-source, this flexibility can be extended further in the future to support even more transports, bindings, and features.

- Transports: Wi-Fi, Ethernet, Serial, Power Line (PLC)
- Bindings: C, C++, Obj-C, Java
- Platforms: RTOS, Arduino, Linux, Android, iOS, Windows, Mac
- Security: peer-to-peer encryption (AES128) and authentication (PSK, ECDSA)

Common language for Internet of Things

In order to fully realize the vision of the Internet of Things, devices and apps need a common way to interact and speak to each other. We believe that common language is the AllJoyn framework: it serves as the glue to allow devices from different companies, running on different operating systems, written with different language bindings to all speak together, and just work.

The AllSeen Alliance, working with the open-source community, is defining and implementing common services and **interfaces** that solves a specific use case, such as **onboarding a new device for the first time**, **sending notifications**, and **controlling a device**. Developers can then take these services, integrate them into their products, and know that they are compatible with other devices and apps in the AllJoyn ecosystem.

Beyond common services and interfaces, an app or device can also implement private interfaces. So, the app can both use common services and interfaces to participate in the larger AllJoyn ecosystem, while at the same time, use the AllJoyn framework to communicate with apps and devices in a private fashion. The AllJoyn framework enables this flexibility.

Optional Cloud

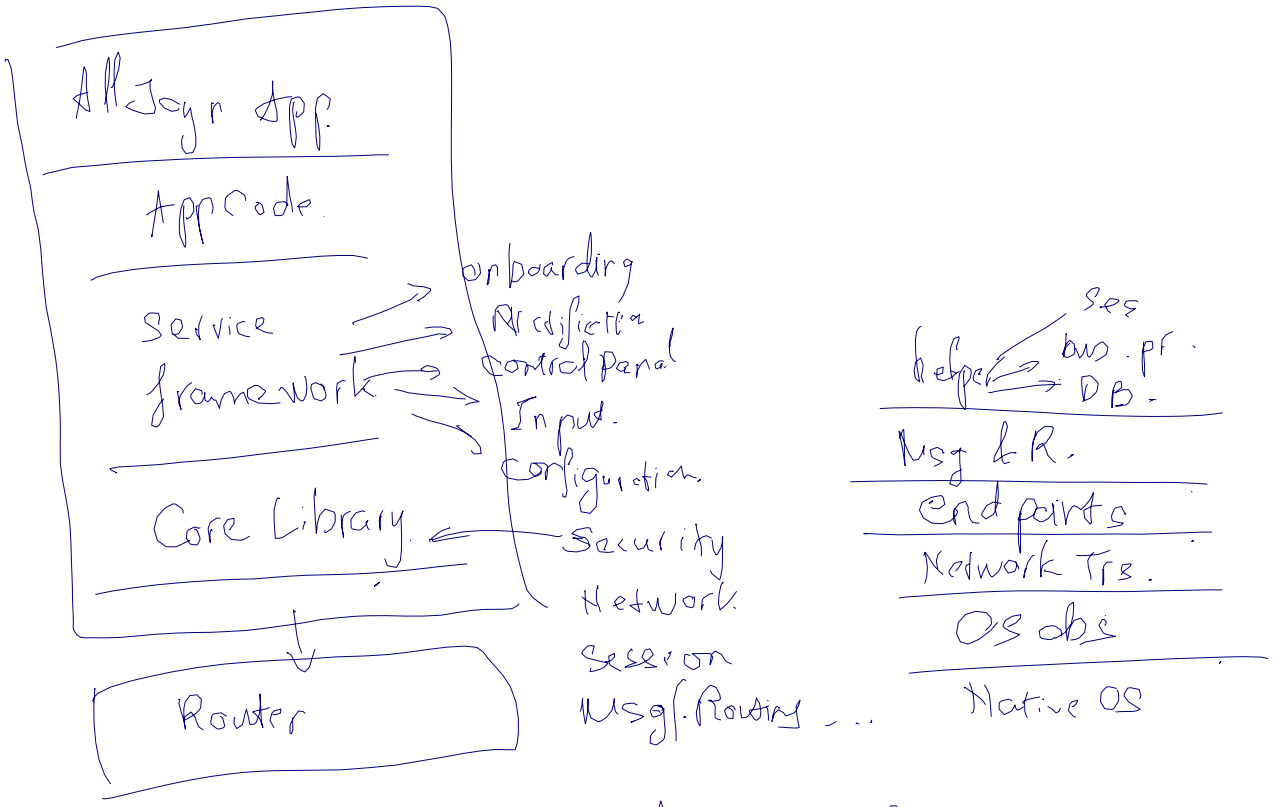
The AllJoyn framework runs on the local network and does not require the cloud to function. Apps and devices talk to each other directly -- fast, efficient, and secure. No need to go out and wait for the cloud when the device is right next to you. And in cases where the cloud is needed, the AllJoyn framework supports that as well through a **Gateway Agent**. One main advantage of this architecture is security: only the Gateway Agent is directly connected to the Internet, reducing the number of devices connected to the Internet, and thus reducing the attack surface.

Momentum

As a collaborative open source project, the AllSeen ecosystem continues to grow and evolve. More common services are being added with each release, including implementation for multiple platforms. There is strong momentum, and with your help, the AllJoyn framework can very well be the common language for the Internet of Things.

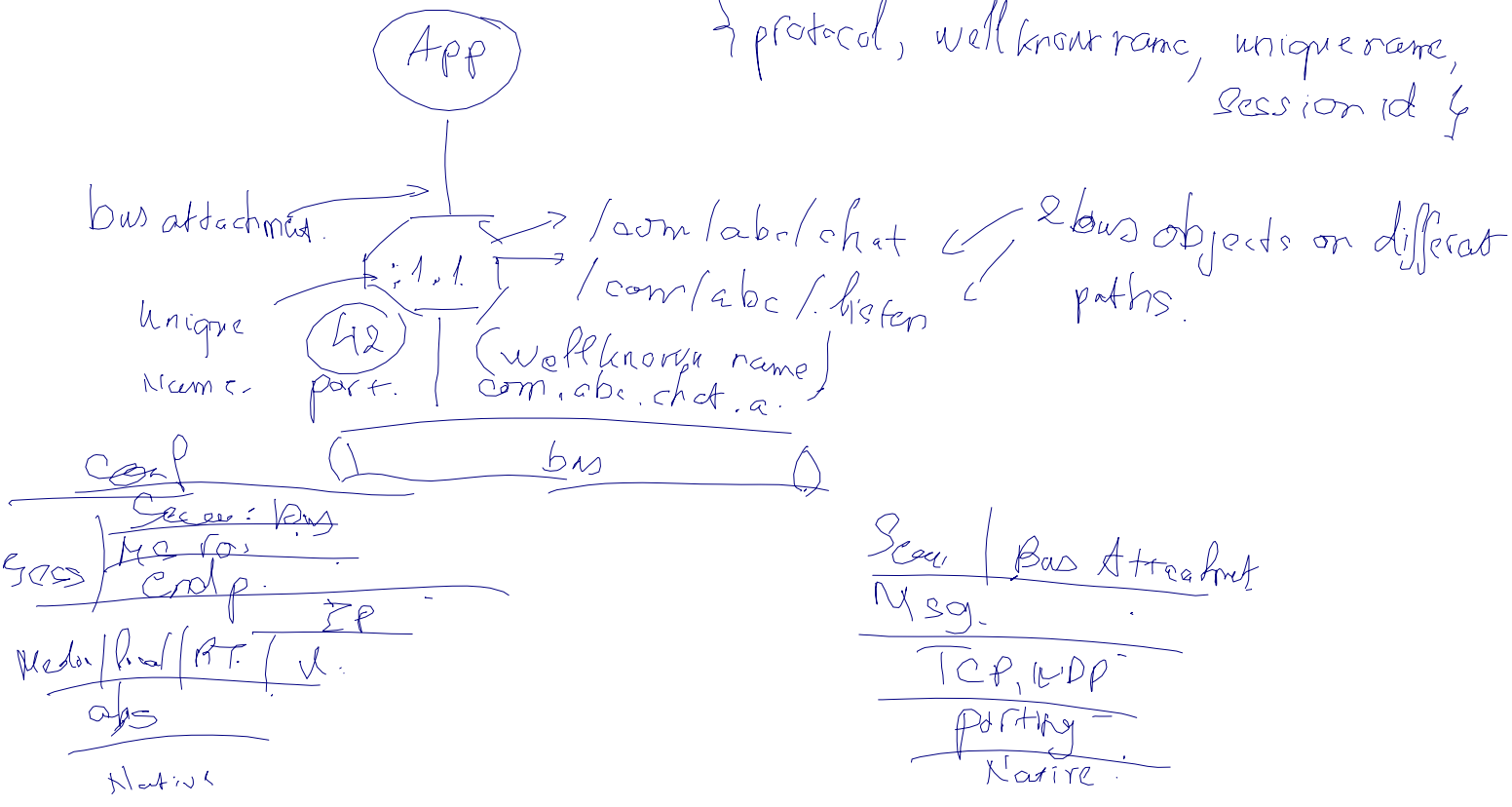
Next steps

Learn more about **use cases**. Then head over to learn about the overall **Architecture**, **Core Framework**, and **Base Services**.



{protocol, well known name, port}

{protocol, well known name, unique name, session id}



ARCHITECTURE

Network Architecture

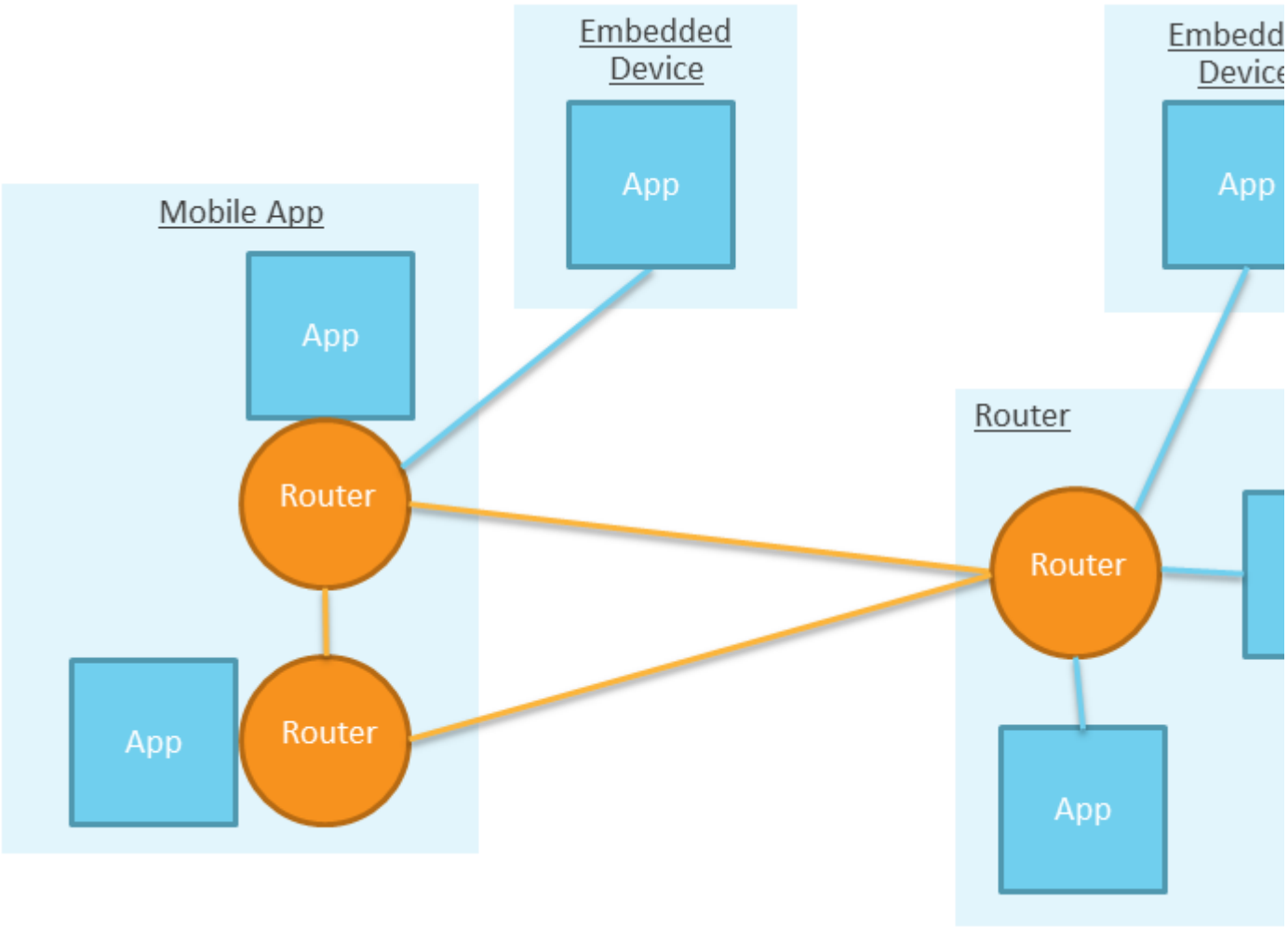
The AllJoyn™ framework runs on the local network. It enables devices and apps to advertise and discover each other. This section explains the network architecture and the relationship between various AllJoyn components.

Apps and Routers

The AllJoyn framework comprises AllJoyn Apps and AllJoyn Routers, or Apps and Routers for short. Apps communicate with Routers and Routers communicate with Apps. Apps can only communicate with other Apps by going through a Router.

Apps and Routers can live on the same physical device, or on different devices. From an AllJoyn perspective, it doesn't matter. In reality, three common topologies exist:

1. An App uses its own Router. In this case, the Router is called a "Bundled Router" as it is bundled with the App. AllJoyn Apps on mobile OSes like Android and iOS and desktop OSes like Mac OS X and Windows generally fall in this group.
2. Multiple Apps on the same device use one Router. In this case, the Router is called a "Standalone Router" and it typically runs in a background/service process. This is common on Linux systems where the AllJoyn Router runs as a daemon process and other AllJoyn apps connect to the Standalone Router. By having multiple apps on the same device use the common AllJoyn Router, the device consumes less overall resources.
3. An App uses a Router on a different device. Embedded devices (which use the Thin variant of the AllJoyn framework, more on this later) typically fall in this camp as the embedded device typically does not have enough CPU and memory to run the AllJoyn router.



Transports

The AllJoyn framework runs on the local network. It currently supports Wi-Fi, Ethernet, serial, and Power Line (PLC), but since the AllJoyn software was written to be transport-agnostic and since the AllJoyn system is an evolving open-source project, support for more transports can be added in the future.

Additionally, bridge software can be created to bridge the AllJoyn framework to other systems like Zigbee, Z-wave, or the cloud. In fact, a Working Group is working on adding a **Gateway Agent** as a standard AllJoyn service.

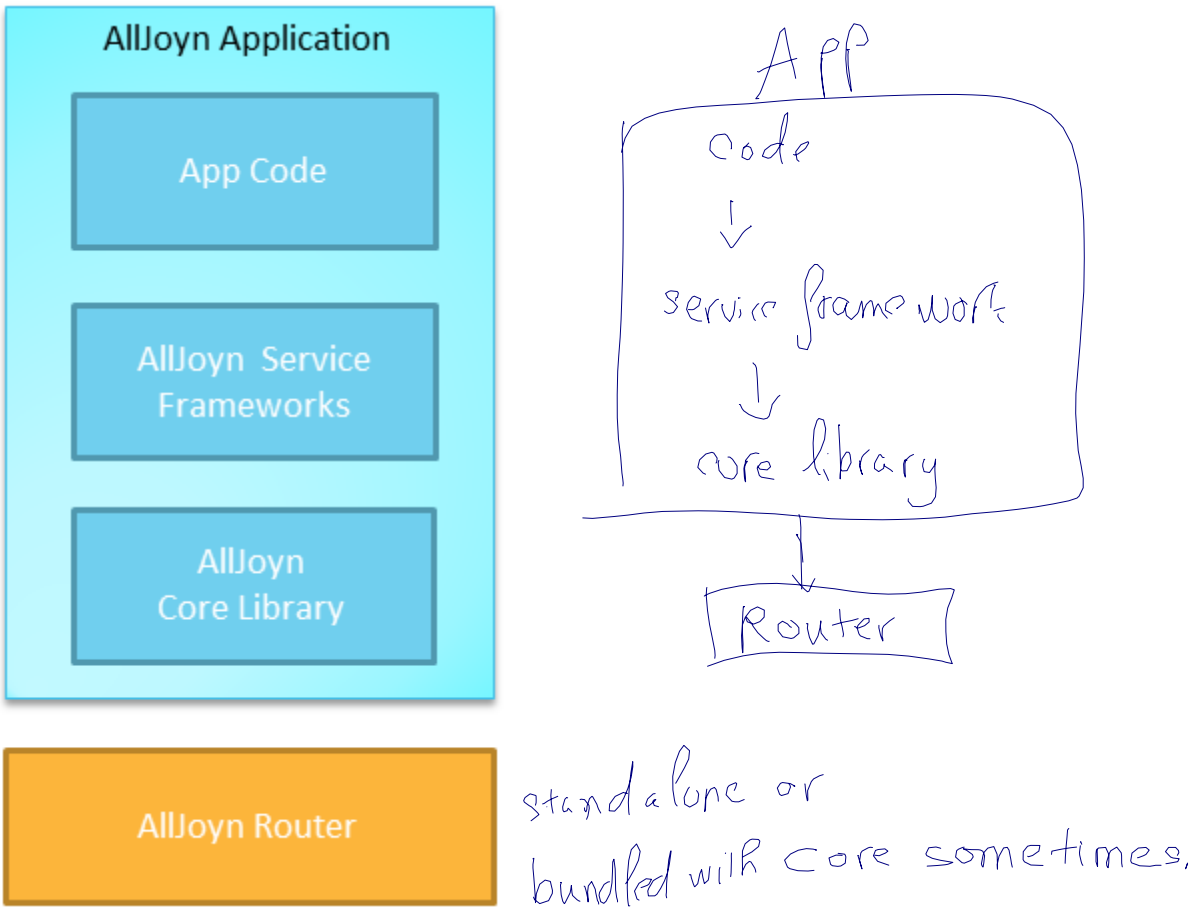
Software Architecture

The AllJoyn network comprises AllJoyn Applications and AllJoyn Routers.

An AllJoyn Application comprises the following components:

- **AllJoyn App Code**
- **AllJoyn Service Frameworks Libraries**
- **AllJoyn Core Library**

  An **AllJoyn Router** can either run as standalone or is sometimes bundled with the AllJoyn Core Library.



AllJoyn Router

The AllJoyn router routes AllJoyn messages between AllJoyn Routers and Applications, including between different transports.

AllJoyn Core Library

The AllJoyn Core Library provides the lowest level set of APIs to interact with the AllJoyn network. It provides direct access to:

- Advertisements and discovery
- Session creation
- Interface defintion of methods, properties, and signals
- Object creation and handling

Developers use these APIs to implement AllJoyn service frameworks, or to implement private interfaces.

Learn more about AllJoyn Core Frameworks.

AllJoyn Service Framework Libraries

The AllJoyn Service Frameworks implement a set of common services, like onboarding, notification, or control panel. By using the common AllJoyn service frameworks, apps and devices can properly interoperate with each other to perform a specific functionality.

Service frameworks are broken out into AllSeen Working Groups:

- **Base Services**
  - **Onboarding**. Provide a consistent way to bring a new device onto the Wi-Fi network.
  - **Configuration**. Allows one to configure certain attributes of an application/device, such as its friendly name.
  - **Notifications**. Allows text-based notifications to be sent and received by devices on the AllJoyn network. Also supports audio and images via URLs.
  - **Control Panel**. Allows devices to advertise a virtual control panel to be controlled remotely.
- **More Service Frameworks**. More service frameworks are actively being developed by the AllSeen Working Groups.

Developers are encouraged to use AllJoyn Service Frameworks where possible. If an existing service is not available, then the developer is encouraged to work with the AllSeen Alliance to create a standard service. In some cases, using private services and intefaces makes the most sense; howerver, those services would not be able to interoperate and take advantage of the larger AllJoyn ecosystem of devices and apps.

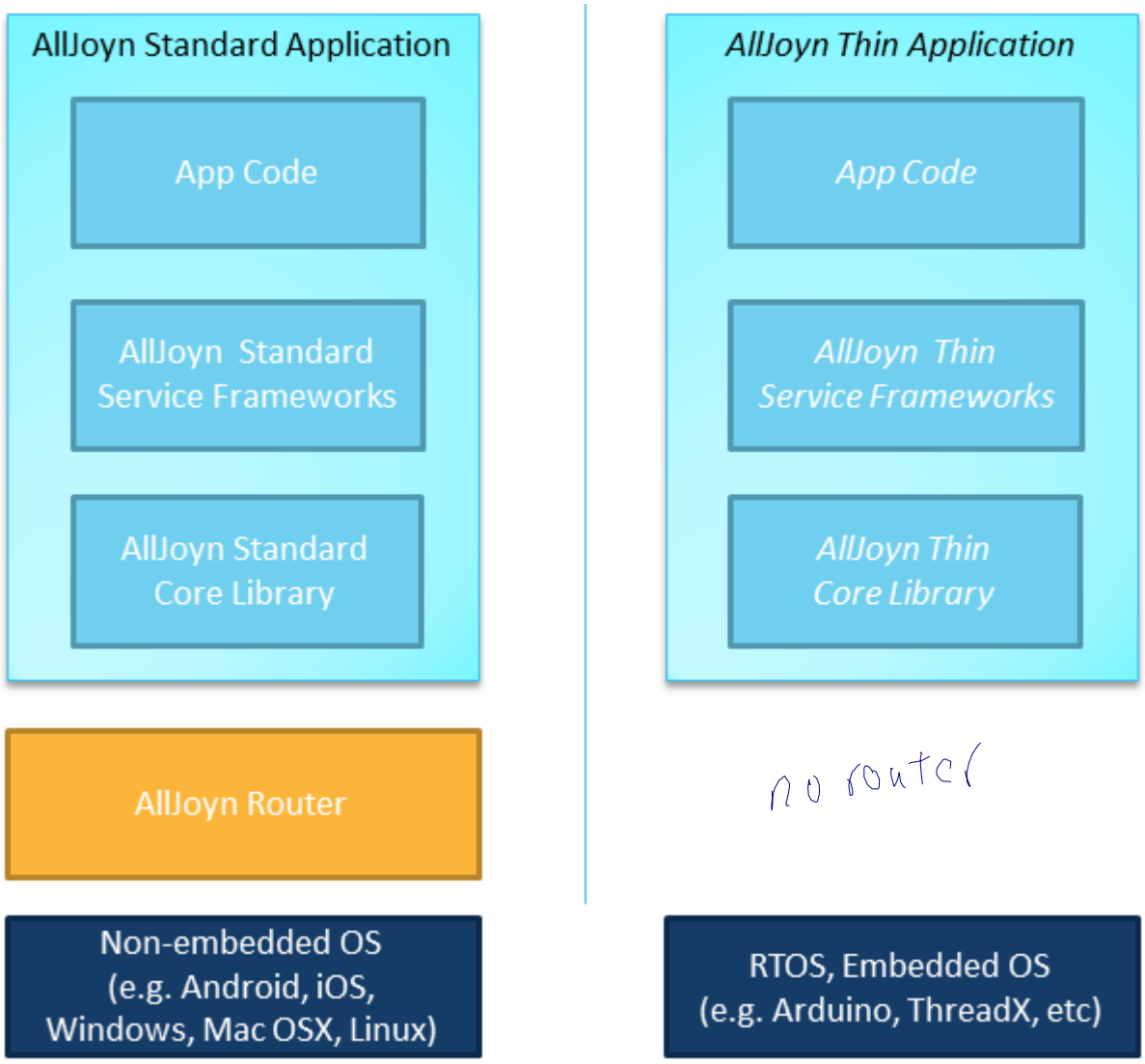
AllJoyn App Code

This is the application logic of the AllJoyn application. It can be programmed to either the AllJoyn Service Frameworks Libraries, which provide higher level functionality, or the AllJoyn Core Library, which provides direct access to the AllJoyn Core APIs.

Thin and Standard

The AllJoyn framework provides two variants:

- Standard. For non-embedded devices, like Android, iOS, Linux.
- Thin. For resource-constrained embedded devices, like Arduino, ThreadX, Linux with limited memory.



Programming Models

Typically, applications will be written using the AllJoyn Service Framework APIs so that the applications can be compatible with devices using the same Service Frameworks. Only by using AllJoyn Service Frameworks developed by AllSeen Working Groups will the application be compatible with other applications and devices in the AllSeen ecosystem.

If an application wishes to implement its own service, it can do so by programming directly to the AllJoyn Core APIs. When doing so, it is recommended to follow the Events and Actions convention to enable ad hoc interactions between other AllJoyn devices.

The application can use both the Service Framework and Core APIs side by side.

Learn more about Events and Actions.

CORE FRAMEWORK

This section describes the AllJoyn™ Core concepts. A base high-level understanding is suggested for anyone developing AllJoyn applications, even if the application is only using AllJoyn Service Frameworks.

App → Bus. → Network/Router

Bus Attachment

AllJoyn Applications use and interact with the AllJoyn network by first instantiating the AllJoyn Bus Attachment object and using that object to connect to the AllJoyn Router.

Advertisement and Discovery

AllJoyn applications can advertise its services via two mechanisms: About Announcements and Well-Known Name. Depending on available transports, the AllJoyn framework will use different mechanisms to ensure that the application can be discovered by other AllJoyn applications. For IP-based transports, mDNS and a combination of multicast and broadcast UDP packets are used.

About Announcements are the recommended mechanism for advertising. It provides a common way for applications to advertise a consistent set of metadata about the application to intersted parties, such as make, model, supported interfaces, a graphical icon, and much more.

Well-Known Name is a more primitive mechanism for applications to announce and discover each other. It is the mechanism that About Announcements use. It is recommended for an application to use About Announcements unless there is a speicifc need for this lower-level functionality.

In both cases, the process of discovery returns a list of AllJoyn applications identified by its UniqueName. This value is used to subsequently create sessions for further communications.

Learn more about About Announcements.

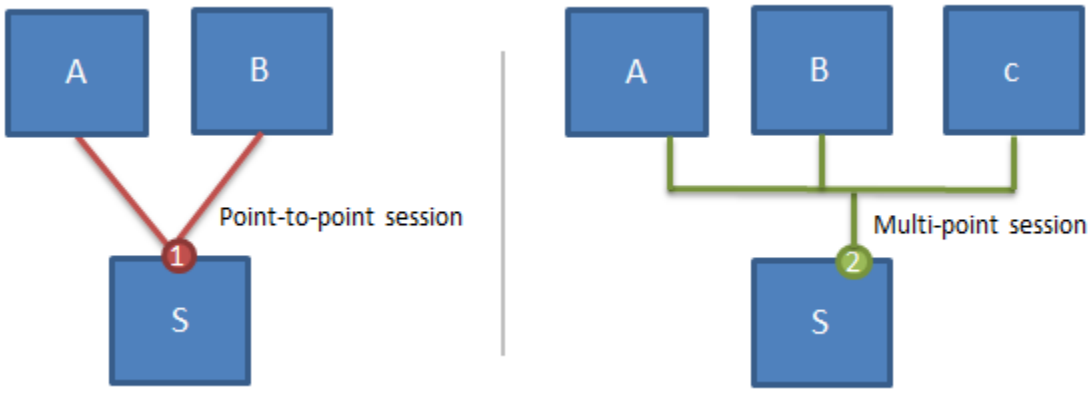
Session and Port

The AllJoyn framework takes care of creating connections between different AllJoyn applications. Typically, an application that is offering a service advertises itself through

About Announcements. A remote application, upon discovering this application (and its UniqueName), can create a session, a process called JoinSession. The application offering the service has the option of accepting or rejecting the JoinSession request.

A session can either be point-to-point or multi-point. Point-to-point sessions allow one-to-one connection, while multi-point allow a group of devices/applications to communicate on the same session.

Sessions are created on a specific port. Different ports allow for a variety of point-to-point and multi-point connection topologies. In the figure below, on the left side, both A and B have a point-to-point connection to S on port 1. On the right side, A, B, C, and S are all connected on a multi-point session on S's port 2.



BusObject

AllJoyn applications communicate with one another via the BusObject abstraction. This abstraction maps well with object-oriented programming concepts where objects are created with well-defined interfaces. Typically, an application that is offering a service creates a BusObject. Remote applications can effectively remotely open the BusObject and call methods on it, similar to a remote procedure call.

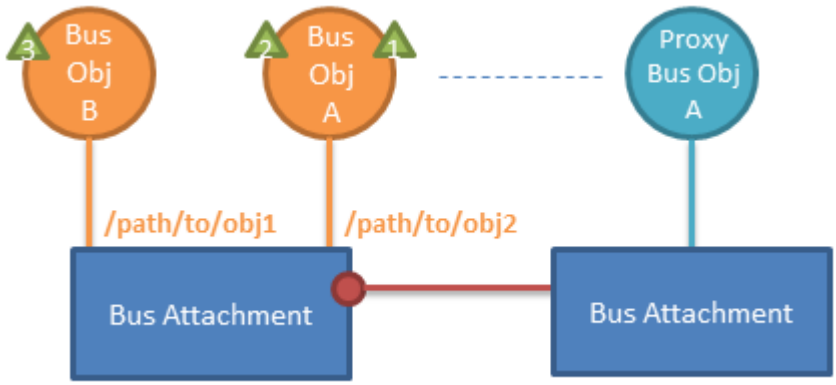
A BusObject can implement a set of interfaces. Each interface clearly defines a set of BusMethods, BusProperties, and BusSignals. BusMethods allow a remote entity to call a method. BusProperties can be get and set. BusSignals are signals emitted from the application offering the service.

A BusObject is attached at a specific bus path. This allows for greater flexibility as the same object can be attached to different bus paths for different purposes. For example, if

an application is implementing a service for a stovetop, the StoveBurner BusObject can be attached to multiple bus paths like '/range/left', and '/range/right' can be created to allow for controls to individual stove burners.

A ProxyBusObject is the object that is created by the remote application to gain access to the BusObject.

Summing up, an application offering a service implements a BusObject to expose access to its services. A remote application creates a session with this application, and connects to its BusObject on a specific object path by creating a ProxyBusObject. Then, it can call BusMethods, access BusProperties, and receive BusSignals.



Sessionless Signal

A Sessionless Signal is a mechanism used to receive signals without having to manually create a session. Under the hood, it uses the Well-Known Name advertise the existence of a new signal. The remote entity automatically creates a transient session, the signal data is received, and the session is torn down. AllJoyn Core APIs provide this abstraction.

Introspection

Introspection is built into the AllJoyn framework. APIs exist to easily introspect a remote AllJoyn application to discover its object paths and objects; its full interface including all methods; and parameters, properties, and signals. Via introspection, one can learn about the remote device and communicate with it without needing prior information about that device.

An application's interfaces and associated methods, signals, and properties are organized and defined by XML. The schema for the introspection XML may be found on the AllSeen Alliance website:

https://allseenalliance.org/schemas/introspect.xsd

Signal & methods

Events and Actions

Events and Actions is a convention for an application to describe its events and actions. By adding simple metadata descriptors to signals and methods, other AllJoyn applications can easily discover what events the application will emit and what actions the application can accept. This allows other applications to dynamically link events and actions together between different devices to create more complex if-this-then-that interactions.

Learn more about Events and Actions.

Security

AllJoyn security occurs at the application level; there is no trust at the device level. Each interface can optionally require security. If required, authentication occurs on demand between the two apps when a method is invoked or to receive a signal. Mulitple authentication mechanisms are supported: PIN code, PSK, or ECDSA (Elliptical Curve Digital Signature Algorithm). Once authenticated, all messages between these two devices are encrypted using AES-128 CMM.

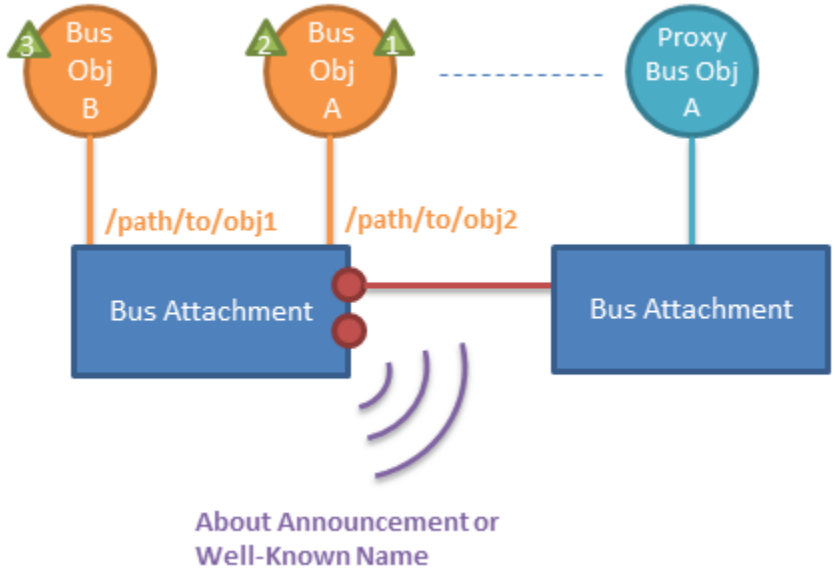
Putting It All Together

An AllJoyn Application interacts with the AllJoyn framework via the Bus Attachment. The application advertises its services via the About Announcement, which lists metadata about the application, including the supportedinterfaces. The UniqueName is returned in discovery to identify the application.

When a remote application discovers an AllJoyn application, it can create a session by connecting to a specific port. Both point-to-point and multi-point sessions are supported. The AllJoyn application has the option of accepting or denying remote connection requests.

Prior to session creation, the application can create any number of bus objects and place them at a specific object path. Each bus object can implement a set of interfaces, defined by a set of methods, properties, and signals.

After the session is created, the remote application will typically communicate with the application by creating a local ProxyBusObject to interact with the BusObject by invoking methods, getting and setting properties, receiving signals.



In many cases, the client-side discovery, session setup and proxy object management follow a simple, common pattern across applications. The Standard Core library offers a convenience API for these cases with the**Observer** class. The Observer class automates About announcement parsing, session management and proxy object creation for the client application.

Learn more

- **Learn more about the AllJoyn Standard Core**
- **Learn more about the AllJoyn Thin Core**
- **Learn more about the low-level details of the AllJoyn system**

ABOUT ANNOUNCEMENT

About Announcements enables a device or app to announce itself on the AllJoyn® network for other devices and apps to discover. The following information is shared:

- App and Device Friendly Names
- Make, Model, Version, Description
- Supported Languages
- App Icon
- Supported objects and interfaces
- Service Port number
- App and Device unique identifiers
  For a complete list, refer to the **Interface Definiton**.

The About feature supports multiple languages, so the client can display the language that is most appropriate for the user. With the About feature, a client can discover devices and apps on the network, get some meta data about the device/app, discover the services it supports, and get an icon to represent the device/app.

Concepts and Terminology

Generally speaking, there are two sides to the About feature:

- About Server. This is the device or app that is announcing itself.
- About Client. This is the device or app that is discovering apps/devices.

How It Works

Here's roughly what happens behind the scenes:

1. An About Server announces itself by sending a sessionless signal including: the session port, list of objects and interfaces; and a subset of the About Announcement information, including App and Device Name, default language, App and Device unique identiers.

2. An About Client discovers the sessionless signal, which includes the information listed above. The client can now display some information about the discovered device/app, App/Device Name and supported services.
3. Optionally, the About Client can connect to the app/device's About Server on the service port to extract more information. Typically, this is done to get the app icon.

Learn More

- **Learn more about the About Interface Definition**
- **Download the SDK**, **build** and **run the sample apps**
- **Learn more about the About APIs**

ABOUT FEATURE INTERFACE DEFINITIONS

Release History

To access a previous version of this document, click the release version link below.

Release version	Date	What changed
14.02	2/28/2014	About interface version 1 was added.
14.06	6/30/2014	No updates.
14.06 Update 1	9/29/2014	Updated the document title and Overview chapter title (changed from Specification to Definition) Added the release version number to the document title for version tracking. Added a note in the Definition Overview chapter to address the AllSeen Alliance Compliance and Certification program. Added a Mandatory column for method and signal parameters to support the AllSeen Alliance Compliance and Certification program.
14.12	12/17/2014	Changed DeviceName from required to not required Additional clarification specifying the AppId must be 128-bit UUID as specified in RFC 4122

Release version	Date	What changed
		Cleanup to make requirements for methods and signals more clear Icon interface was added. The icon interface has been part of AllJoyn™ and the About Feature since 14.02; however, the interface definition documentation was not added until 14.12.

Definition Overview

The About interface is to be implemented by an application on a target device. This interface allows the app to advertise itself so other apps can discover it. The following figure illustrates the relationship between a client app and a service app.

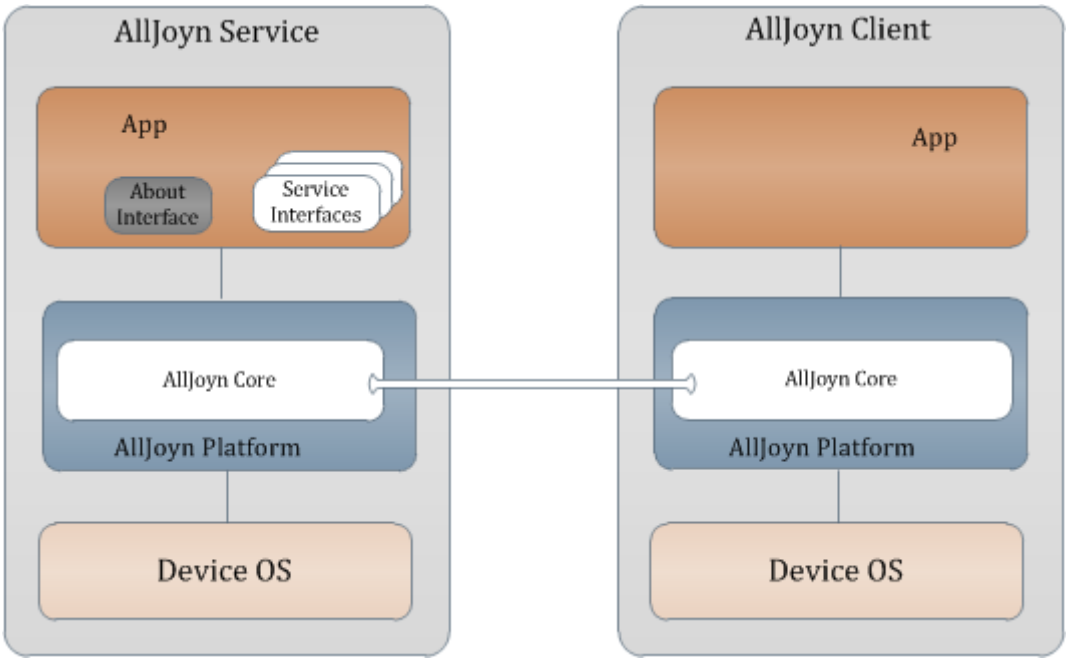


Figure: About feature architecture within the AllJoyn™ framework

NOTE: All methods and signals are considered mandatory to support the AllSeen Alliance Compliance and Certification program.

Discovery

A client can discover the app via an announcement which is a sessionless signal containing the basic app information like app name, device name, manufacturer, and

model number. The announcement also contains the list of object paths and service framework interfaces to allow the client to determine whether the app provides functionality of interest.

In addition to the sessionless announcement, the About interface also provides the on-demand method calls to retrieve all the available metadata about the app that are not published in the announcement.

Discovery Call Flows

Typical discovery flow

The following figure illustrates a typical call flow for a client to discover a service app. The client merely relies on the sessionless announcement to decide whether to connect to the service app to use its service framework offering.

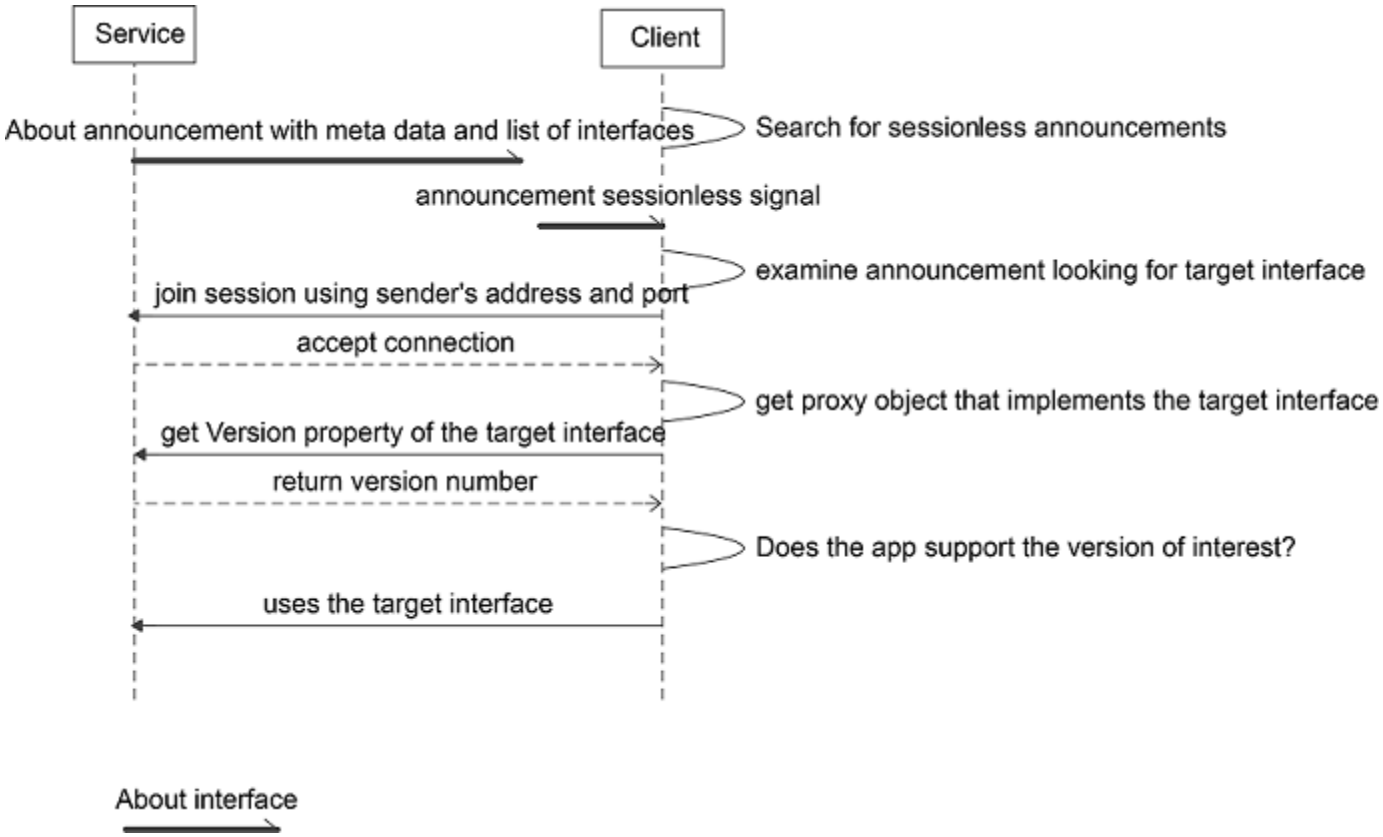


Figure: Typical discovery flow (client discovers a service app)

Nontypical discovery flow

The following figure illustrates a call flow for a client to discover a service app and make a request for more detailed information.

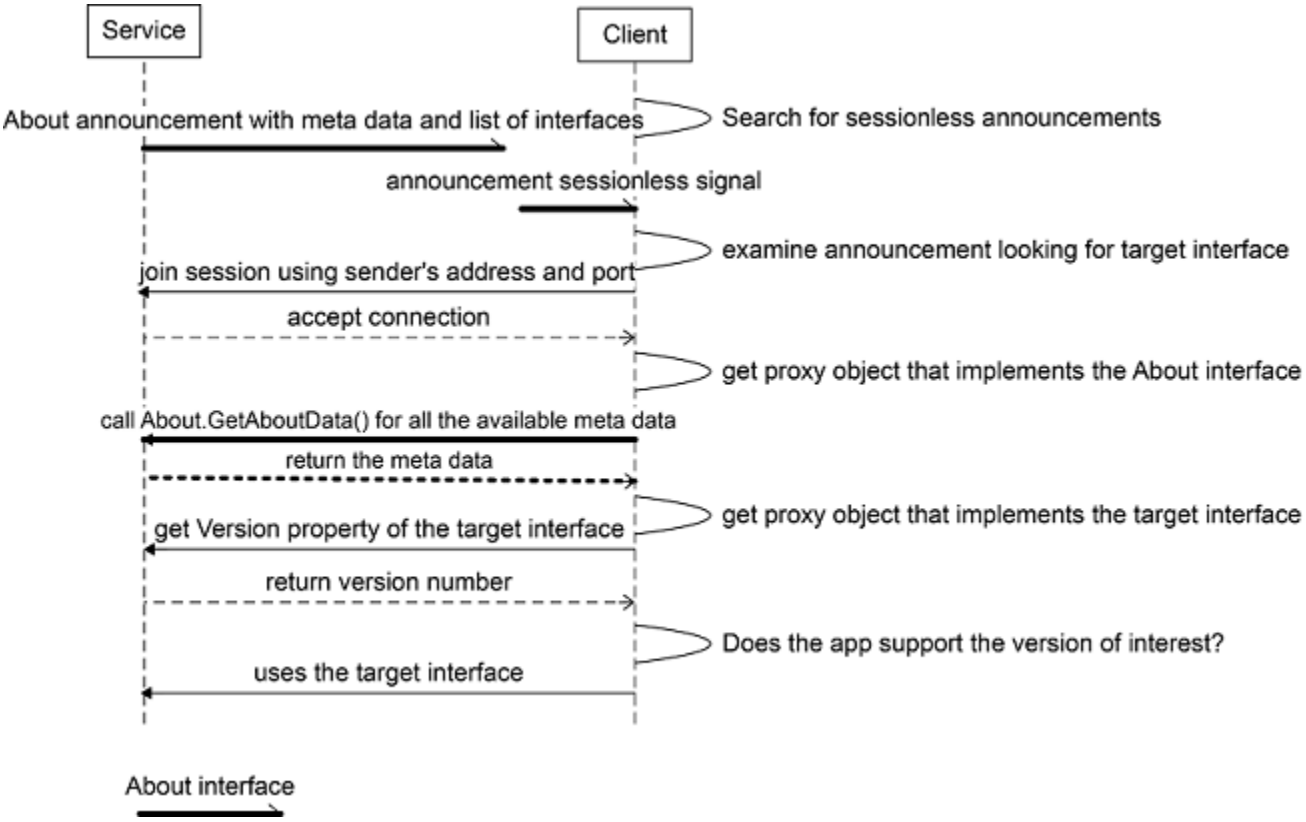


Figure: Nontypical discovery call flow

Error Handling

The method calls in the About interface will use the AllJoyn error message handling feature (ER_BUS_REPLY_IS_ERROR_MESSAGE) to set the error name and error message.

Error name	Error message
org.alljoyn.Error.LanguageNotSupported	The language specified is not supported

About Interface

Interface name	Version	Secured	Object path
`org.alljoyn.About`	1	no	`/About`

Properties

Property name	Signature	List of values	Read/Write	Description
Version	`q`	Positive integers	Read Only	Interface version number

Methods

The following methods are exposed by a BusObject that implements the `org.alljoyn.About` interface.

`a{sv} GetAboutData('s')`

Message arguments

Argument	Parameter name	Signature	List of values	Description
0	`languageTag`	`s`	IETF language tags specified by**RFC 5646**.	The desired language.

Reply arguments

Argument	Parameter name	Return signature	Description
0	`AboutData`	`a{sv}`	A dictionary of the available metadata fields. If language tag is not specified (i.e., ""), metadata fields based on default language are returned.

Error reply

Error	Description
`org.alljoyn.Error.LanguageNotSupported`	Returned if a language tag is not supported

Description

Retrieve the list of available AboutData fields based on the language tag. see **About data interface fields**

About data interface fields

The following table lists the names of the metadata fields. The fields with a yes value in the Announced column will also be published via the Announce signal. See **Signals** for more information.

Field name	Mandatory	Announced	Localized	Signature	Description
`AppId`	yes	yes	no	`ay`	A 128-bit globally unique identifier for the application. The AppId shall be a universally unique identifier as specified in**RFC 4122**.
`DefaultLanguage`	yes	yes	no	`s`	The default language supported by the device. Specified as an IETF language tag listed in**RFC 5646**.
`DeviceName`	no	yes	yes	`s`	Name of the device set by platform-specific means (such as Linux and Android).
`DeviceId`	yes	yes	no	`s`	Device identifier set by platform-specific means.
`AppName`	yes	yes	yes	`s`	Application name assigned by the app manufacturer

Field name	Mandatory	Announced	Localized	Signature	Description
					(developer or the OEM).
`Manufacturer`	yes	yes	yes	`s`	The manufacturer's name of the app.
`ModelNumber`	yes	yes	no	`s`	The app model number.
`SupportedLanguages`	yes	no	no	`as`	List of supported languages.
`Description`	yes	no	yes	`s`	Detailed description expressed in language tags as in RFC 5646.
`DateOfManufacture`	no	no	no	`s`	Date of manufacture using format YYYY-MM-DD (known as XML DateTime format).
`SoftwareVersion`	yes	no	no	`s`	Software version of the app.
`AJSoftwareVersion`	yes	no	no	`s`	Current version of the AllJoyn SDK used by the application.
`HardwareVersion`	no	no	no	`s`	Hardware version of the device on which the app is running.

Field name	Mandatory	Announced	Localized	Signature	Description
`SupportUrl`	no	no	no	`s`	Support URL (populated by the manufacturer).

`a(oas) GetObjectDescription()`

Message arguments

None.

Reply arguments

Argument	Parameter name	Return signature	Description
0	`objectDescription`	`a(oas)`	Return the array of object paths and the list of supported interfaces provided by each object.

Description

Retrieve the object paths and the list of all interfaces implemented by each of objects.

Signals

The following signals are emitted by a BusObject that implements the `org.alljoyn.About` interface.

`Announce('qqa(oas)a{sv}')`

Announce signal is a Sessionless signal

Message arguments

Argument	Parameter name	Signature	List of values	Description
0	`version`	`q`	positive	Version number of the About interface.
1	`port`	`q`	positive	Session port the app will listen on incoming sessions.

Argument	Parameter name	Signature	List of values	Description
2	`objectDescription`	`a(oas)`	Populated based on announced interfaces	Array of object paths and the list of supported interfaces provided by each object.
3	`aboutData`	`a{sv}`	array of key/value pairs	All the fields listed in **About data interface fields** with a yes value in the Announced column are provided in this signal.

AllJoyn Introspection XML

```
<node name="/About" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="http://www.allseenalliance.org/schemas/introspect.xsd"
>
   <interface name="org.alljoyn.About">
      <property name="Version" type="q" access="read"/>
      <method name="GetAboutData">
         <arg name="languageTag" type="s" direction="in"/>
         <arg name="aboutData" type="a{sv}" direction="out"/>
      </method>
      <method name="GetObjectDescription">
         <arg name="objectDescription" type="a(oas)" direction="out"/>
      </method>
      <signal name="Announce">
         <arg name="version" type="q"/>
         <arg name="port" type="q"/>
         <arg name="objectDescription" type="a(oas)"/>
         <arg name="metaData" type="a{sv}"/>
      </signal>
   </interface>
</node>
```

Icon Interface

Interface name	Version	Secured	Object path
`org.alljoyn.Icon`	1	no	`/About/DeviceIcon`

Properties

Property name	Signature	List of values	Read/Write	Description
`Version`	`q`	Positive integers	Read Only	Interface version number
`MimeType`	`s`	The Mime type corresponding to the icon's binary content	Read Only	Mime type for the icon
`Size`	`u`	The size in bytes of the icons binary content	Read Only	Size of the Icon

Methods

The following methods are exposed by a BusObject that implements the `org.alljoyn.Icon` interface.

`s GetUrl()`

Message arguments

None.

Reply arguments

Argument	Parameter name	Return signature	Description
0	`url`	`s`	The URL if the icon is hosted on the cloud

Description

Retrieve the URL of the icon if the icon is hosted on the cloud.

`ay GetContent()`

Argument	Parameter name	Return signature	Description
0	`content`	`ay`	The binary content for the icon

Signals

None.

AllJoyn Introspection XML

```
<node name="/About/DeviceIcon"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="http://www.alljoyn.org/schemas/introspect.xsd">
    <interface name="org.alljoyn.Icon">
        <property name="Version" type="q" access="read"/>
        <property name="MimeType" type="s" access="read"/>
        <property name="Size" type="u" access="read"/>
        <method name="GetUrl">
            <arg type="s" direction="out"/>
        </method>
        <method name="GetContent">
            <arg type="ay" direction="out"/>
        </method>
    </interface>
</node>
```

EVENTS AND ACTIONS

The Events and Actions feature is a generic mechanism that simple interoperability between AllJoyn apps and devices. It enables apps and devices to send events that can be easily discovered and received by other apps and devices. Similarly, apps and devices can offer actions for other apps and devices to discover and invoke. For example, a proximity sensor can emit an event when someone walks by and a lamp can accept an action to turn on the light. By being able to discover these events and actions, one can create an app to turn on the light when the sensor trips. This simple mechanism, with its discoverability, enables apps to create dynamic ad hoc interactions.

Events and Actions build on top of the AllJoyn® Core. An event is merely a signal within an AllJoyn interface with an associated human readable `description` that explains what the event is. Similarly, an action is just a method with a `description` in an AllJoyn interface. Standard AllJoyn Core APIs are used to send and receive events and to invoke and process actions.
Here is an example of what an interface with a `description` looks like:

```
<interface name="com.example.LightBulb">
  <method name="ToggleSwitch">
     <description>Toggle light switch</description>
  </method>
  <signal name="LightOn" sessionless="true">
     <description>The light has been turned on</description>
  </signal>
</interface>
```

Since all AllJoyn interfaces are introspectable, so are events and actions.
The `description` tag, which supports multiple languages, is provided to give the user some information about what the event and action is.

When an event is connected to an action a sentence is formed. Using the above example a connection of the two creates "The light has been turned on, Toggle the light switch".

Like with all AllJoyn interfaces, events and actions can require security, thus limiting those who can receive events and call actions.

Learn more

See the **Events and Actions API Guide** to more learn how to add Events and Actions into your application.

ROUTING NODE CONFIGURATION FILE

The routing node (RN) configuration file is an XML file that controls the behavior of the RN and sets certain variables, such as timers, connection limits, and use case characteristics.

An example config file may be found in the AllJoyn source distribution in `../alljoyn_core/router/test/conf/sample.conf`

Note that this sample configuration file would not be useful in practice. It is merely intended to provide an example of how various elements and attributes are formatted.

XML Schema

An XML schema, which may be used to validate config files, may be found on the AllSeen Alliance website:

https://allseenalliance.org/schemas/busconfig.xsd
The schema is also available in the source code distribution at: `../alljoyn_core/docs/busconfig.xsd`
The schema may be referenced in config files by including the following attributes in the `busconfig` (i.e., root) element:

```
<busconfig
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="https://allseenalliance.org/schemas/busconfig.xsd">
```

Default Configuration

Bundled Routing Node

The bundled routing node consists of a routing node and a Standard Client application that run in a single process.

As of v15.09, a user may compile AllJoyn with the scons variable `TEST_CONFIG` set to an absolute or relative path to a config file. However, as suggested by the scons variable name, this method is intended for debug and test purposes.
If the user does not provide a config file, then the bundled RN relies on a hardcoded configuration contained in the file `BundledRouter.cc`.

Note that all config settings that apply to the standalone RN also apply to the bundled RN.

Bundled RN hardcoded configuration:

```
<busconfig>
    <type>alljoyn_bundled</type>
    <listen>tcp:iface=*,port=0</listen>
    <listen>udp:iface=*,port=0</listen>
    <limit name="auth_timeout">20000</limit>
    <limit name="max_incomplete_connections">4</limit>
    <limit name="max_completed_connections">16</limit>
    <limit name="max_remote_clients_tcp">8</limit>
    <limit name="max_remote_clients_udp">8</limit>
    <property name="router_power_source">Battery powered and chargeable</property>
    <property name="router_mobility">Intermediate mobility</property>
    <property name="router_availability">3-6 hr</property>
    <property name="router_node_connection">Wireless</property>
</busconfig>
```

Standalone Routing Node

The standalone RN runs as a compiled application named `alljoyn-daemon` on Linux. If the user does not specify a separate configuration file when alljoyn-daemon is started, or the `--internal` switch is used, then the RN will use a configuration that is hardcoded in the source.
Note that the standalone RN is also available as a service on Windows 10. The Windows 10 service uses a hard-coded configuration and does not accept any parameters or switches. The examples in this document that describe passing parameters or options to `alljoyn-daemon` are referring to the Linux implementation.

Example (Linux): start routing node using internal config

```
./alljoyn-daemon

    --OR--

./alljoyn-daemon --internal
```

Standalone RN hardcoded configuration (Linux):

```
<busconfig>
    <type>alljoyn</type>
    <listen>unix:abstract=alljoyn</listen>
```

```
    <listen>tcp:iface=*,port=9955</listen>
    <listen>udp:iface=*,port=9955</listen>
    <limit name="auth_timeout">20000</limit>
    <limit name="max_incomplete_connections">16</limit>
    <limit name="max_completed_connections">32</limit>
    <limit name="max_remote_clients_tcp">0</limit>
    <limit name="max_remote_clients_udp">0</limit>
</busconfig>
```

Using an External Config File

As previously mentioned, a user-defined, external configuration file may be passed to a standalone RN. In order to start the standalone RN with an external configuration file, enter the following command (Linux):

```
./alljoyn-daemon --config-file=./my_rn_config.xml
```

Minimal Config File

A minimal config file requires the root element (`<busconfig>`) and a single `<listen>` element.

For example:

```
<busconfig>
    <listen>tcp:iface=*,port=9955</listen>
</busconfig>
```

This will result in a valid config file, but with very rudimentary functionality. In fact, this configuration file would prevent the connection of any leaf nodes, since it does not override the default values of 0 for`max_remote_clients_tcp` and `max_remote_clients_udp` (i.e., `<limit>` attributes). Therefore, it would only be able to connect to other routing nodes.

Review of Config File Elements and Attributes

The default configuration settings should be suitable for most use cases. However, some applications may benefit from using different settings. Therefore, the information below is intended to provide some guidance in using these settings.

The following is an alphabetical list of top-level XML elements (i.e., immediate child elements of the root element `<busconfig>`) available for use in the config file, along with a description of their meaning and application. Any child elements will be described below their parent element. Likewise for attributes.

`<auth>`

Not used by AllJoyn. Included for backward compatibility. Including this element in the config file will have no effect.

`<flag>`

Note: The content of the flag element may be either 'true' or 'false'. All flags default to 'false'

name attribute (required)

name	description	default	Notes
ns_enable_v1	enable legacy name service if true	true	Used for debugging. Do not use in deployed production applications.
ns_disable_ipv4	disable IPv4 multicast if true	false	Used for debugging. Do not use in deployed production applications.
ns_disable_ipv6	disable IPv6 multicast if true	false	Used for debugging. Do not use in deployed production applications.
ns_disable_directed_broadcast	disable subnet broadcast if true	false	Used for debugging. Do not use in deployed production applications.

Examples

1. `<flag name="ns_enable_v1">false</flag>`
2. `<flag name="ns_disable_ipv6">true</flag>`

`<fork>`

The fork element is empty. It does not contain any content, child elements, or attributes.

If present in the config file, the routing node (daemon) will run in the background.

Note that `<fork>` must be enabled in order for `<pidfile>` to be effective.
On Linux, the config file setting for `<fork>` may be overridden from the command line, by using the `--fork` or `--no-fork` options.

Examples

1. `<fork/>`

   `<include>`

   Specify the path to another configuration file, whose contents will be included at the point where the`<include>` element is inserted.
   The specified path may be absolute or relative. If the path is relative, then it is relative to the configuration file using the `<include>`.
   In addition, the path must be to a file with the file extension `.conf`.

   ignore_missing attribute (optional)

   May be set to "yes" or "no". The default is "no".

ignore_missing	description
no (default)	a fatal error will occur if the file doesn't exist
yes	silently ignore if the file doesn't exist

   Examples

1. `<include>../configs/my_config.conf</include>`
2. `<include ignore_missing="yes">my_config.conf</include>`
3. `<include ignore_missing="no">/home/bob/configs/my_config.conf</include>`

   `<includedir>`

   Similar to `<include>`, except that, instead of specifying the path to a file, specify the path to a directory containing configuration files. All files with the `.conf` file extension will be included.

   Note that the order in which the files in the directory are included is undefined and, therefore, users should avoid any config file include-order dependency when using this element.

   The specified path may be absolute or relative. If the path is relative, then it is relative to the configuration file using the `<includedir>`

ignore_missing attribute (optional)

May be set to "yes" or "no". The default is "no".

ignore_missing	description
no (default)	a fatal error will occur if the directory doesn't exist
yes	silently ignore if the directory doesn't exist

Examples

1. `<includedir>../configs</includedir>`
2. `<includedir ignore_missing="yes">configs</includedir>`
3. `<includedir ignore_missing="no">/home/bob/configs</includedir>`

`<limit>`

Sets variables for various timeouts and connection limits to an unsigned, 32-bit integer.

name attribute (required)

name	description	default
auth_timeout	The maximum amount of time that incoming connections are allowed to complete authentication. Beyond this limit, incoming connections may be aborted.	20000 (ms)
session_setup_timeout	The maximum amount of time that incoming connections are allowed to set up session routes. Beyond this limit, incoming connections may be disconnected.	30000 (ms)
max_incomplete_connections	The maximum number of incoming connections that can be in the process of authenticating. If starting to authenticate a new connection would mean exceeding this number, then the new connection will be dropped.	10 (connections)
max_completed_connections	The maximum number of connections (inbound and outbound) allowed for each tranport. This means that the total number of completed connections allowed for TCP and UDP is twice this value.	50 (connections)

name	description	default
	If starting to process a new connection would mean exceeding this number, then the new connection will be dropped.	
max_remote_clients_tcp	The maximum number of remote clients using TCP. Note: this value *must* be overridden when using an external configuration file in order to enable TCP connections to leaf nodes. The default internal configuration file already overrides this value. Note that this setting may be useful for applications that have a specific requirement to prevent leaf nodes from connecting; however, for generic RNs, this value should not be zero.	0 (connections)
max_remote_clients_udp	The maximum number of remote clients using UDP. Note: this value *must* be overridden when using an external configuration file in order to enable UDP connections to leaf nodes. The default internal configuration file already overrides this value. Note that this setting may be useful for applications that have a specific requirement to prevent leaf nodes from connecting; however, for generic RNs, this value should not be zero.	0 (connections)

Test hooks *only*

Many of the limits found in the source code are hooks for testing AllJoyn and are not useful for application development or end-products. For informational purposes, these are listed below, however, please do not use them for production code, as they may have undesirable or unpredictable behavior.

DO NOT USE	
slap_min_idle_timeout	udp_timewait

DO NOT USE	
slap_max_idle_timeout	udp_segbmax
slap_default_idle_timeout	udp_segmax
slap_max_probe_timeout	udp_fast_retransmit_ack_counter
slap_default_probe_timeout	sls_backoff
udp_connect_timeout	sls_backoff_linear
udp_connect_retries	sls_backoff_exponential
udp_initial_data_timeout	sls_backoff_max
udp_total_data_retry_timeout	sls_preferred_transports
udp_min_data_retries	tcp_default_probe_timeout
udp_persist_interval	tcp_min_idle_timeout
udp_total_app_timeout	tcp_max_idle_timeout
udp_link_timeout	tcp_default_idle_timeout
udp_keepalive_retries	tcp_max_probe_timeout
udp_delayed_ack_timeout	dt_min_idle_timeout
dt_max_idle_timeout	dt_default_idle_timeout
dt_max_probe_timeout	dt_default_probe_timeout

Examples

1. `<limit name="auth_timeout">20000</limit>`
2. `<limit name="max_incomplete_connections">16</limit>`

`<listen>`

Identifies an address on which the bus attachment should listen for incoming connections. This address must be a valid URI that begins with an AllJoyn transport name, followed by appropriate parameters and options.

A configuration file must contain at least one `<listen>` element. In addition, a configuration file may contain multiple `<listen>` elements, including for the same transport, as long as the specified transport and other parameters are valid and supported by the host platform.

Duplicate `<listen>` elements will be ignored (with a warning message).
Invalid `<listen>` elements will result in a fatal error, unless there is a valid `<listen>` element available, in which case the invalid`<listen>` element will be ignored (with a warning message).

Descripton and Format

Protocol or Transport Name	Description	Format	Example
tcp	AllJoyn over TCP	`tcp:(iface=<interface>` | `addr=<IPv4 address),port=<port number>`	`tcp:iface=lo,port=9955`
udp	AllJoyn over UDP	`udp:(iface=<interface>` | `addr=<IPv4 address),port=<port number>`	`udp:iface=*,port=9955`
unix	Unix domain socket. Applicable only for standalone routing nodes on POSIX platforms.	`unix:(abstract`&#124 `path)=<named socket>`	`unix:abstract=alljoyn`
slap	Serial Line over AllJoyn Protocol. Used to provide an interface between an AllJoyn embedded platform and a POSIX-based routing node.	`slap:type=<com interface type>,dev=<serial port>,baud=<baud rate>`	`slap:type=uart,dev=/dev/tty`

Protocol or Transport Name	Description	Format	Example
npipe	Windows 10 named pipe protocol	`npipe:` (Note: no options or other parameters available)	`npipe:`

Format options

Option Name	Content	Use Wildcard?	Notes
iface	Valid network interface name (see definition of `ifName` in RFC 2863) on the device that is hosting the RN. For example, `eth0` on Linux, or `Ethernet_32803` on Windows.	Yes	`iface` and `addr` are mutually exclusive `iface` is preferred over `addr`
addr	Valid IPv4 address	Yes	`iface` and `addr` are mutually exclusive `addr` primarily exists for backward compatibility. Prefer `iface`, instead.
port	Valid TCP or UDP port number	No	`port=0` indicates that the transport will use an ephemeral port.
abstract	Named socket	No	Linux only. `unix:abstract=alljoyn` is the default because applications look for the "alljoyn" socket name by default
path	POSIX named socket	No	The specified path must refer to an existing named socket.
type	Communications interface type.	No	At this time, only `uart` is supported.
dev	Serial port	No	POSIX platform specific. Only tested on Linux.

Option Name	Content	Use Wildcard?	Notes
baud	Baud rate of the specified serial port.	No	

`<pidfile>`

Records the routing node process ID (PID) to the specified file. If the file does not exist, it will be created. If the file does exist, it will be over-written.

IMPORTANT NOTE: `<pidfile>` is ONLY effective if `<fork>` is also specified. Alternatively, the routing node may be started with the `--fork` option (Linux).

`<policy>`

Defines a security policy to be applied to a particular set of connections to the bus. A policy is made up of`<allow>` and `<deny>` elements. Policies are analogous to a firewall in that they allow expected traffic and prevent unexpected traffic.

Policies applied later will override those applied earlier, when the policies overlap.

Attributes

The `<policy>` element must include one of the following attributes, which are mutually exclusive. A fatal error will be generated if more than one attribute is present.

Attribute	Contents	Priority
context	"default" OR "mandatory"	`context="default"` is the lowest priority. Policy rules in this category apply if none of the rules match from the other categories. `context="mandatory"` is the highest priority. Policy rules in this category override rules on all other categories.
user	username or userid	`user="uid"` is higher priority than context="default" and group="gid", but lower priority than context="mandatory". Policy rules in this category override those in context="default" and group="gid".
group	group name or gid	`group="gid"` is higher priority than context="default" and lower priority than user="uid". Policy rules in this category override those in context="default".

<allow> and <deny>

A <policy> must contain at least one <allow> or <deny> element. However, a single <policy>with only an <allow> element is not meaningful for AllJoyn, since all traffic is allowed by default. Instead, an<allow> element primarily functions as an exception to previous <deny> elements.

Attributes for <allow> and <deny> elements

Attribute	Contents	Example
user	username or userid	user="joes"
group	group name or gid	group="enemies"
own	bus name	own="com.companyA.ProductA"
own_prefix	bus name prefix	own_prefix="com.companyA" would match com.companyA.productA and com.companyA.productI
send_error / receive_error	Enable or disable specific error	receive_error="org.alljoyn.Error.Foo"

Attribute	Contents	Example
	messages for sending or receiving	
send_interface / receive_interface	Enable or disable specific interfaces for sending or receiving	`receive_interface="com.companyA.InterfaceB"`
send_member / receive_member	Enable or disable signals or methods for sending or receiving	`receive_member="some_signal_name"`
send_path / receive_path	Enable or disable specific object paths for sending or receiving	`send_path="/org/alljoyn/lighting"`
send_path_prefix / receive_path_prefix	Enable or disable all object paths that match a particular prefix	For example `send_path_prefix="/org"` would match `/org/alljoyn`, `/org/CompanyA`, etc.
send_group / receive_group	Group name or GID. Rule will match if the sending or receiving group matches the specified group.	`send_group="mycompany"`
send_user	Username or uderid. Rule will match if the receiving endpoint matches the specified user.	`send_user="joes"`
receive_user	Username or uderid. Rule will match if the sending endpoint	`receive_user="beth"`

Attribute	Contents	Example
	matches the specified user.	
send_same_user / receive_same_user	Specify whether the same user credentials are associated with each connected application, or not. Must be "true" or "false".	For example `send_same_user="false"`
receive_sender / send_destination	Enable or disable sending to, or receiving from, specific bus names	`receive_sender="com.companyA.productB"`
send_type / receive_type	Enable or disable message types. Must be "method_call", "method_return", "signal", or "error"	`send_type="method_call"`

Example

```
<policy context="default">
  <deny user="*"/>
  <deny own="*"/>

  <deny send_type="method_call"/>
  <allow send_type="signal"/>
  <allow send_type="method_return"/>
  <allow send_type="error"/>

  <allow send_destination="org.freedesktop.DBus"/>
  <allow receive_sender="org.freedesktop.DBus"/>

  <allow send_interface="org.alljoyn.Bus.Peer.Session"/>

  <allow user="jethro"/>

</policy>
```

```
<policy user="jethro">
  <allow send_type="method_call"/>
  <allow send_type="signal"/>
  <allow send_type="method_return"/>
  <allow send_type="error"/>

  <allow own_prefix="test"/>
</policy>

<policy user="joe">
   <deny send_type="method_call" send_user="beth"/>
   <deny receive_type="signal" receive_user="bob"/>
</policy>
```

`<property>`

Define characteristics related to router node selection and nameservice.

name attribute (required)

Name	Description	Default	Notes
router_node_connection	Used for routing node selection. One of the following: "access point", "wired", "wireless"	"wireless"	If one of the enumerated values is not used, then the value will be ignored, a warning message will be logged, and the default value will be used
router_availability	Used for routing node selection. One of the following: "0-3 hr", "3-6 hr", "6-9 hr", "9-12 hr", "12-15 hr", "15-18 hr", "18-21 hr", "21-24 hr"	"3-6 hr"	If one of the enumerated values is not used, then the value will be ignored, a warning message will be logged, and the default value will be used

Name	Description	Default	Notes
router_mobility	Used for routing node selection. One of the following: "always stationary", "low mobility", "intermediate mobility", "high mobility"	"intermediate mobility"	If one of the enumerated values is not used, then the value will be ignored, a warning message will be logged, and the default value will be used
router_power_source	Used for routing node selection. One of the following: "always ac powered", "battery powered and chargeable", "battery powered and not chargeable"	"Battery powered and chargeable"	If one of the enumerated values is not used, then the value will be ignored, a warning message will be logged, and the default value will be used
router_advertisement_prefix	Used by thin core applications (TCA) to discover routing nodes.	`"org.alljoyn.BusNode."`	In order to maximize availability to TCA, generic routing nodes should not modify this value. However, it may be useful when deploying custom systems that need to limit the TCA connecting to the routing node due to specific performance constraints.

Example

```
<property name="router_power_source">Battery powered and chargeable</property>
<property name="router_mobility">Intermediate mobility</property>
<property name="router_availability">3-6 hr</property>
<property name="router_node_connection">Wireless</property>
```

`<syslog>`

Has no value or attributes. If present, indicates that AllJoyn Daemon should send log messages to `syslog`.

Example

`<syslog/>`

`<type>`

Not used by AllJoyn. Included for backward compatibility. Including this element in the config file will have no effect.

`<user>`

Indicates that AllJoyn daemon should run as the indicated user instead of root. Note that the indicated user must be known to the system.

Example

`<user>barts</user>`

ALLJOYN® STANDARD CORE

Overview

The AllJoyn framework is an open-source software system that provides an environment for distributed applications running across different device classes with an emphasis on mobility, security, and dynamic configuration. The AllJoyn system handles the hard problems inherent in heterogeneous distributed systems and addresses the unique issues that arise when mobility enters the equation. This leaves application developers free to concentrate on the core problems of the application they are building.

The AllJoyn framework is "platform-neutral", meaning it was designed to be as independent as possible of the specifics of the operating system, hardware, and software of the device on which it is running. fact, the AllJoyn framework was developed to run on Microsoft Windows, Linux, Android, iOS, OS X, OpenWRT, and as a Unity plug-in for the Unity game development ecosystem.

The AllJoyn framework is designed with the concept of proximity and mobility always in mind. In a mobile environment, devices will constantly be entering and leaving the proximity of other devices, and underlying network capacities can be changing as well.

The AllJoyn SDKs are available at (**http://www.allseenalliance.org**).

The types of applications that will use the AllJoyn framework are limited only by the imagination of developers. Extending social networking is one example. A user could define a profile with likes and interests. Upon entering a location, the AllJoyn-enabled handset would immediately discover other nearby peers with similar interests, create a communication network between the peer devices, allow them to communicate, and exchange information.

The majority of handsets today have Wi-Fi integrated, so if two users walk into a home or office that has a Wi-Fi hotspot, their devices can connect to the underlying access point and transparently take advantage of the additional network capacity. Additionally, their devices can locate other devices in the proximity (defined by the Wi-Fi coverage footprint), can discover additional services on the other devices, and use those services, if desired. Further, it is possible to leverage a mixed topology connection such that a device taking advantage of the AllJoyn Thin LIbrary can be designated to use Bluetooth as a transport. As such, once connected to a device that runs the AllJoyn framework, the device can interact with the applications on the Wi-Fi devices.

Enabling real-time multi-player gaming is another example of how the AllJoyn framework might be used. For example, a multi-user game can be accomplished using different device classes such as laptops, tablets, and handsets; and different underlying network technologies such as Wi-Fi. The details of the infrastructure management are all handled by the AllJoyn framework, allowing the game author to focus on the design and implementation of the game, rather than dealing with the complexities of the peer-to-peer networking.

As the AllJoyn ecosystem expands, one can imagine any number of applications. For example:

- Create a playlist consisting of music, and stream the songs to an AllJoyn-enabled car stereo system, or store them on a home stereo (subject to digital rights management)
- Sync recent photos or other media to an AllJoyn-enabled digital picture frame or television upon returning home from an event or trip
- Control home appliances such as televisions, DVRs, or game consoles
- Interact and share content with laptops and desktop computers in the area
- Engage in project collaboration between colleagues and students in enterprise and educational settings
- Provide proximity-based services like distributing coupons or vcards

Benefits of the AllJoyn Framework

As mentioned, the AllJoyn framework is a platform-neutral system that is designed to simplify proximity networking across heterogeneous distributed mobile systems.

Heterogeneous in this case means not only different devices, but different kinds of devices (e.g., PCs, handsets, tablets, consumer electronics devices) running on different operating systems, using different communication technologies.

Open source

The AllJoyn framework is being developed as an open source project. This means that all of the AllJoyn codebase is available for inspection, and developers are encouraged to contribute additions and enhancements. If the AllJoyn framework is missing a feature, you

can contribute. If you run into a snag using the AllJoyn framework, or have a technical question, other participants in the open source community are ready and willing to provide help and guidance. The AllJoyn codebase is available at (**http://www.allseenalliance.org**).

Operating system independence

The AllJoyn framework provides an abstraction layer allowing AllJoyn framework code and its applications to run on multiple OS platforms. As of this writing, the AllJoyn framework supports most standard Linux distributions including Ubuntu, and runs on Android 2.3 (Gingerbread) and later smartphones and tablets. The AllJoyn framework code also runs and is tested and validated on commonly available versions of the Microsoft Windows operating system including Windows XP, Windows 7, Windows RT, and Windows 8. Additionally, the AllJoyn framework code runs on Apple operating systems iOS and OS X, on embedded operating systems such as OpenWRT, and works with the Unity game development ecosystem.

Language independence

Currently, developers may create applications using C++,Java, C#, JavaScript, and Objective-C.

Physical network and protocol independence

There are many technologies available to networked devices. The AllJoyn framework provides an abstraction layer that defines clean interfaces to the underlying network stacks and makes it relatively easy for a competent software engineer to add new networking implementations.

For example, as of this writing, the Wi-Fi Alliance has recently released a specification for Wi-Fi Direct, which will allow for point-to-point Wi-Fi connectivity. A networking module for Wi-Fi Direct is actively being developed that will transparently add Wi-Fi Direct and its pre-association discovery mechanisms to the available networking options for AllJoyn developers.

Dynamic configuration

Often, as a mobile device makes its way through the various locations it encounters during its lifetime, associations with networks may come and go. This means that IP (Internet

Protocol) addresses may change, network interfaces may become unusable, and services may be transitory.

The AllJoyn framework notices when old services are lost and new services appear, and forms new associations when required. The AllJoyn framework is primed and ready as an application layer for Wi-Fi Hotspot 2.0 - a technology that aims to bring the roaming transparency of cell phones and cell towers to Wi-Fi hotspots.

Service advertisement and discovery

Whenever devices need to communicate, there must be some form of service advertisement and discovery. In the old days of static networks, human administrators made explicit arrangements to enable devices to communicate. More recently, the concepts of zero configuration networks have been popularized, especially with Apple Bonjour, and Microsoft Universal Plug and Play. We also see existing technology-specific discovery mechanisms available such the Bluetooth Service Discovery Protocol and emerging mechanisms such as the Wi-Fi Direct P2P Discovery specification. The AllJoyn framework provides a service advertisement and discovery abstraction that simplifies the process of locating and consuming services.

Security

The natural model for security in distributed applications is application-to-application. Unfortunately, in many cases, the network security model does not match this natural arrangement. For example, the Bluetooth protocol requires pairing between devices. Using this approach, once devices are paired, all applications on both devices are authorized. This may not be desirable when considering something more capable than a Bluetooth headset. For example, if two laptops are connected over Bluetooth, a much finer granularity is necessary. The AllJoyn framework is designed to provide extensive support for complex security models such as this, with an emphasis on application-to-application communication.

Object model and remote method invocation

The AllJoyn framework utilizes an easy-to-understand object model and Remote Method Invocation (RMI) mechanism. The AllJoyn model re-implements the wire protocol set forth by the D-Bus specification and extends the D-Bus wire protocol to support distributed devices.

Software componentry

Along with a standard object model and wire protocol comes the ability to standardize various interfaces into components. In much the same way that a Java Interface declaration provides a specification to interact with a local instantiation of an implementation, the AllJoyn object model provides a language-independent specification to interact with a remote implementation.

Using a specification, many interface implementations can be considered, thereby enabling standard definitions for application communication. This is the enabling technology for software componentry. Software components are at the heart of many modern systems, and are especially visible in systems such as Android, which defines four primary component types as the only way to participate in the Android Application Framework; or in Microsoft systems which use descendants of the Component Object Model (COM) system.

We expect that a rich "sea" of interface definitions will emerge in order to enable the scenarios described in**Overview**. The AllJoyn project expects to work with users to define and publish standard interfaces and support the sharing of implementations.

Conceptual Overview

The AllJoyn framework contains a number of abstractions used to help understand and relate the various pieces. There is only a small number of key abstractions that one must know in order to understand AllJoyn-based systems.

This section provides a high-level view of the AllJoyn framework to provide a foundation for follow-on documents such as the detailed API documentation.

Remote Method Invocation

Distributed systems are groups of autonomous computers communicating over some form of network in order to achieve a common goal. Consider the ability of a program running in one address space on one machine to call a procedure located in another address space on a physically separate machine as if it were local. This is usually accomplished through Remote Procedure Call (RPC) or, if object-oriented concepts are in play, RMI or Remote Invocation (RI).

The basic model in an RPC exchange involves a ***client***, which is the caller of the RPC, and a server (called a service in the AllJoyn model), which actually executes the desired remote procedure. The caller executes a client stub that looks just like a local procedure on the local system. The client stub packages up the parameters of its procedure (called marshaling or serializing the parameters) into some form of message and then calls in to the RPC system to arrange delivery of the message over some standard transport mechanism such as the Transmission Control Protocol (TCP). At the remote machine, there is a corresponding RPC system running, which unmarshals (deserializes) the parameters and delivers the message to a server stub that arranges to execute the desired procedure. If the called procedure needs to return any information, a similar process is used to convey the return values back to the client stub, which in turn returns them to the original caller.

Note that it is not required that a given process only implement a client personality or a service personality. If two or more processes implement the same client and service aspects,
they are considered peers. In many cases, AllJoyn applications will implement similar functionality and be considered peers. The AllJoyn framework supports both classic client and service functions and also peer-to-peer networking.

AllJoyn bus

The most basic abstraction of the AllJoyn system is the AllJoyn bus. It provides a fast, lightweight way to move marshaled messages around the distributed system. One can view the AllJoyn bus as a kind of "freeway" over which those messages flow. The following figure shows what an instance of an AllJoyn bus on a single device might look like, conceptually.

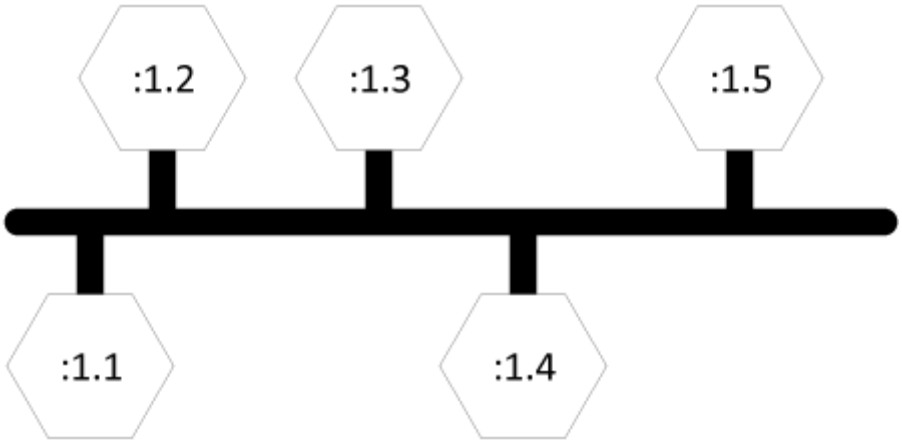


Figure: Prototypical AllJoyn bus

Points about the prototypical AllJoyn bus are detailed below.

- The bus itself is shown as the thick horizontal dark line. The vertical lines can be thought of as "exits" and are the sources and/or destinations of messages that are flowing over the bus.
- The connections to the bus are depicted as hexagons. Just as the exits on a freeway are typically assigned numbers, each connection is assigned a unique connection name. A simplified form of the connection name is used here for clarity.
- In many cases, the connections to the bus can be thought of as co-resident with processes. Therefore, the unique connection name `:1.1` may be assigned to a connection in a process running some instance of an application, and the unique connection name `:1.4` may be assigned to a connection in a process running an instance of some other application. The goal of the AllJoyn bus is to allow the two applications to communicate without having to deal with the details of the underlying mechanisms. One of the connections can be thought of as the client stub, and the other side can fulfill the duties of the service stub.

The prototypical AllJoyn bus figure shows an instance of an AllJoyn bus and illustrates how a software bus can provide interprocess communication between components attached to the bus. The AllJoyn bus is typically extended across devices as shown in the following figure. A communication link between the segment of the logical bus residing on the Smartphone and the components residing on the Linux host is formed when required by the components.

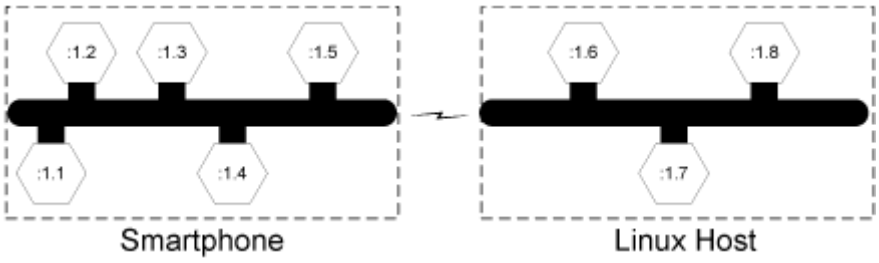


Figure: Device-to-device communication handled by the AllJoyn framework

The management of this communication link is handled by the AllJoyn system and may be formed using a number of underlying technologies such as Wi-Fi or Wi-Fi Direct. There

may be multiple devices involved in hosting the AllJoyn bus, but this is transparent to the users of the distributed bus. To a component on the bus, a distributed AllJoyn system looks like a bus that is local to the device.

The following figure shows how the distributed bus may appear to a user of the bus. A component (for example, the Smartphone connection labeled `:1.1`) can make a procedure call to the component labeled `:1.7` on the Linux host without having to worry about the location of that component.

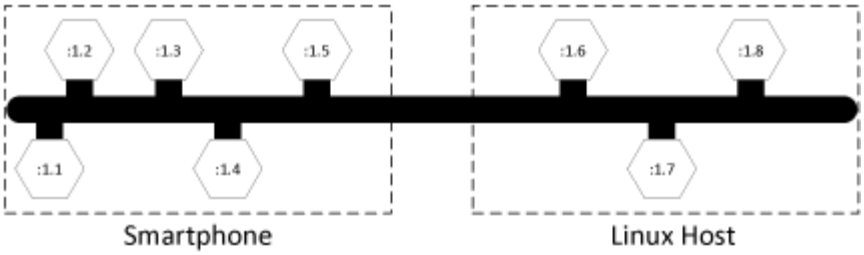


Figure: A distributed AllJoyn bus appears as a local bus

Bus router

The device-to-device communication figure illustrates that the logical distributed bus is actually split up into a number of segments, each running on a different device. The AllJoyn functionality that implements these logical bus segments is called an AllJoyn router.

The term daemon is commonly used in Unix-derived systems to describe programs that run to provide some needed functionality to the computer system. On a Linux system instead of saying daemon we call it the standalone router. In Windows systems, the term service is more typically used, however we refer to it as the AllJoyn router.

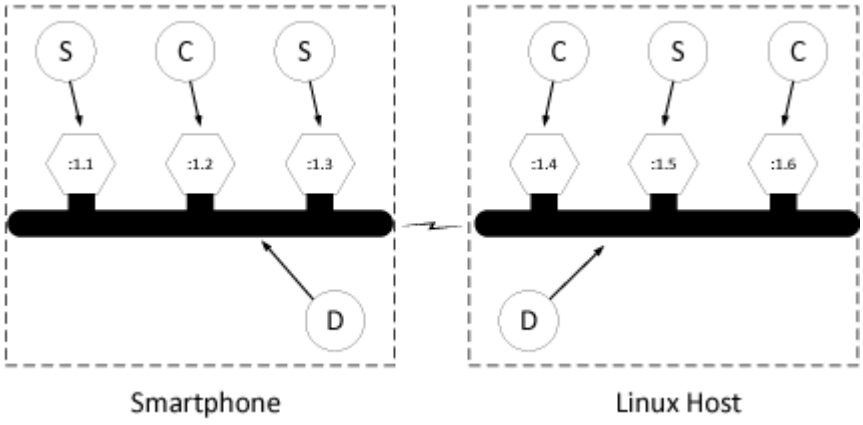


Figure: Relating bubble diagrams to the bus

In order to visualize the AllJoyn router, it is useful to create a bubble diagram. Consider two AllJoyn bus segments, one residing on a Smartphone and one on a Linux Host, as shown in previous figure. The connections to the bus are labeled as clients (C) and services (S) using the sense of clients and services in the RMI model. The AllJoyn router that implements the core of the distributed bus is labeled (D). The components of the previous figure are typically translated into the illustration shown in the folliwing figure.

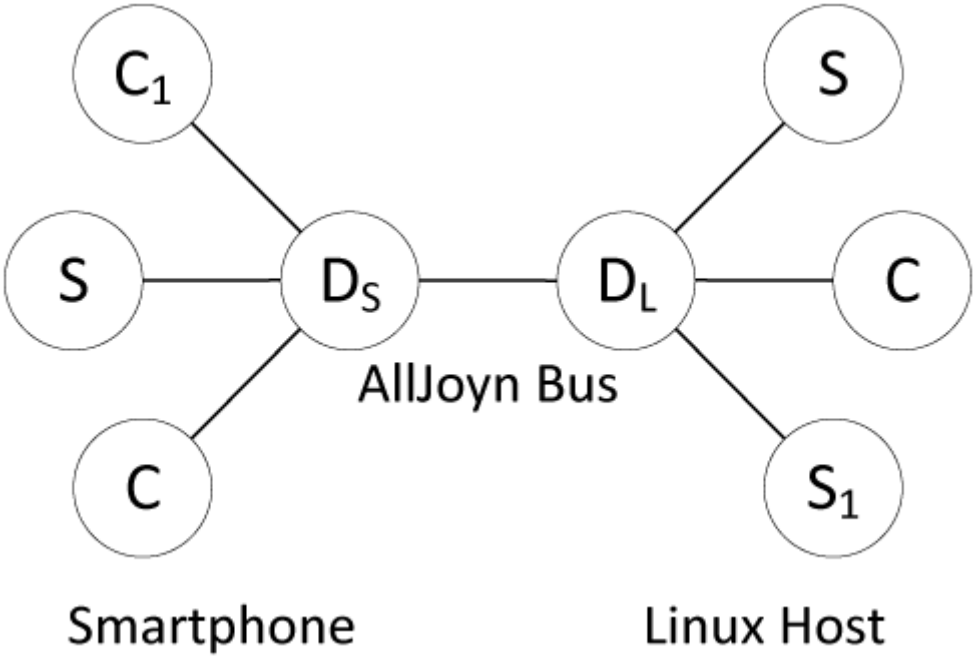


Figure: AllJoyn bubble diagrams

The bubbles can be viewed as computer processes running on a distributed system. The two client (C) and the service (S) processes on the left are running on the Smartphone. These three processes communicate with an AllJoyn router running on the Smartphone which implements the local segment of the distributed AllJoyn bus. On the right side, there is a router which implements the local segment of the AllJoyn bus on the Linux Host.

These two routing nodes coordinate the message flow across the logical bus, which appears as a single entity to the connections, as shown in the distributed AllJoyn bus figure. Similar to the configuration on the Smartphone, there are two service components and a client component on the Linux host.

In this configuration, client component C1 can make remote method calls to service component S1 as if it were a local object. Parameters are marshaled at the source and routed off of the local bus segment by the router residing on the Smartphone. The marshaled parameters are sent over the network link (transparently from the perspective of the client) to the routing node on the Linux host. The AllJoyn router running on the Linux host determines that the destination is S1 and arranges to have the parameters unmarshaled and the remote method invoked on the service. If return values are expected, the process is reversed to communicate the return values back to the client.

Since the standalone routers are running in a background process and the clients and services are running in separate processes, there must be a "representative" of the routers in each of those separate processes. The AllJoyn framework calls these representatives bus attachments.

Bus attachments

Every connection to the AllJoyn bus is mediated by a specific AllJoyn component called a bus attachment. A bus attachment lives in each process that has a need to connect to the AllJoyn software bus.

An analogy is often drawn between hardware and software when discussing software components. One can think of a local segment of a distributed AllJoyn bus in much the same way as one thinks of the hardware backplane bus in a desktop computer. The hardware bus itself moves electronic messages and has attachment points called connectors into which one plugs cards. The analogous function of the connector in the AllJoyn framework is the bus attachment.

An AllJoyn bus attachment is a local language-specific object that represents the distributed AllJoyn bus to a client, service, or peer. For example, there is an implementation of the bus attachment functionality provided for users of the C++ language, and there is an implementation of the same bus attachment functionality provided for users of the Java language. As the AllJoyn framework adds language bindings, more of these language-specific implementations will become available.

Bus methods bus properties and bus signals

The AllJoyn framework is fundamentally an object-oriented system. In object-oriented systems, one speaks of invoking methods on objects (thus the term Remote Method

Invocation when speaking of distributed systems). Objects in the object-oriented programming sense have members. Classically, these are object methods or properties, which are known as BusMethods and BusProperties in the AllJoyn framework. The AllJoyn framework also has the concept of a BusSignal, which is an asynchronous notification of some event or state change in an object.

In order to transparently arrange for communication between clients, services, and peers, there must be some specification of the parameter ordering for bus methods and bus signals, and some form of type information for bus properties. In computer science, the description or definition of the types of the inputs and outputs of a method or signal is called the type signature.

Type signatures are defined by character strings. Type signatures can describe character strings, all of the basic number types available in most programming languages, and composite types such as arrays and structures built up from these basic types. The specific assignment and use of type signatures is beyond the scope of this introduction, but the type signature of a bus method, signal, or property conveys to the underlying AllJoyn system how to convert the passed parameters and return values to and from the marshaled representation over the bus.

Bus interfaces

In most object-oriented programming systems, collections of methods or properties are composed into groups that have some inherent common relationship. A unified declaration of this collection of functions is called an interface. The interface serves as a contract between an entity implementing the interface specification and the outside world. As such, interfaces are candidates for standardization by appropriate standards bodies. Specifications for numerous interfaces for services ranging from telephony to media player control can be found on various web sites. Interfaces specified this way are described in XML as per the D-Bus specification.

An interface definition collects a group of bus methods, bus signals, and bus properties along with their associated type signatures into a named group. In practice, interfaces are implemented by client, service, or peer processes. If a given named interface is implemented, there is an implicit contract between the implementation and the outside world that the interface supports all of the bus methods, bus signals, and bus properties of the interface.

Interface names typically take the form of a reversed domain name. For example, there are a number of standard interfaces that the AllJoyn framework implements. One of the AllJoyn standard interfaces is the `org.alljoyn.Bus` interface which routers implement and which provides some of the basic functionality for bus attachments.
It is worth noting that the interface name is simply a string in a relatively free-form namespace and that other namespaces may have a similar look. The interface name string serves a specific function that should not be confused with other similar strings, in particular bus names. For example, `org.alljoyn.sample.chat` may be a bus name which is the constant, unchanging name that a client will search for. It may also be the case that `org.alljoyn.sample.chat` is the interface name that defines the methods, signals and properties available in a bus object associated with a bus attachment of the specified bus name. The existence of an interface with the given interface name is implied by the existence of the bus name; however, they are two completely different things that can sometimes look exactly the same.

Bus objects and object paths

The bus interface provides a standard way to declare an interface that works across the distributed system. The bus object provides the scaffolding into which an implementation of a given interface specification may be placed. Bus objects live in bus attachments and serve as endpoints of communication.

Since there may be multiple implementations of a specific interface in any particular bus attachment, there must be additional structure to differentiate these interface implementations. This is provided by an object path.

Just as an interface name is a string that lives in an interface namespace, the object path lives in a namespace. The namespace is structured as a tree, and the model for thinking about paths is a directory tree in a filesystem. In fact, the path separator in an object path is the slash character (/), just as in a Unix filesystem. Since bus objects are implementations of bus interfaces, object paths might follow the naming convention of the corresponding interface. In the case of an interface defining a disk controller interface (for example, `org.freedesktop.DeviceKit.Disks`), one could imagine a case where multiple implementations of this interface were described by the following object paths corresponding to an implementation of the interface for two separate physical disks in a system:

```
/org/freedesktop/DeviceKit/Disks/sda1
```

```
/org/freedesktop/DeviceKit/Disks/sda2
```

Proxy bus object

Bus objects on an AllJoyn bus are accessed through proxies. A proxy is a local representation of a remote object that is accessed through the bus. Proxy is a common term that is not specific to the AllJoyn system, however you will often encounter the term ProxyBusObject in the context of the AllJoyn framework to indicate the specific nature of the proxy - that it is a local proxy for a remotely located bus object.

The ProxyBusObject is the portion of the low-level AllJoyn code that enables the basic functionality of an object proxy.

Typically, the goal of an RMI system is to provide a proxy that implements an interface which looks just like that of the remote object that will be called. The proxy object implements the same interface as the remote object, but drives the process of marshaling the parameters and sending the data to the service.

In the AllJoyn framework, the client and service software, often through specific programming language bindings, provides the actual user-level proxy object. This user-level proxy object uses the capabilities of the AllJoyn proxy bus object to accomplish its goal of local/remote transparency.

Bus names

A connection on the AllJoyn bus acting as a service provides implementations of interfaces described by interface names. The interface implementations are organized into a tree of bus objects in the service. Clients wishing to consume the services do so via proxy objects, which use lower-level AllJoyn proxy bus objects to arrange for delivery of bus method-, bus signal- and bus property-related information across the logical AllJoyn bus.

In order to complete the addressing picture of the bus, connections to the bus must have unique names. The AllJoyn system assigns a unique temporary bus name to each bus attachment. However, this unique name is autogenerated each time the service connects to the bus and is therefore unsuitable for use as a persistent service identifier. There must be a consistent and persistent way to refer to services attached to the bus. These persistent names are referred to as *well-known names*.

Just as one might refer to a host system on the Internet by a domain name that does not change over time (e.g., quicinc.com), one refers to a functional unit on the AllJoyn bus by its well-known bus name. Just as interface names appear to be reversed domain names, bus names have the same appearance. Note that this is the source of some confusion, since interface names and well-known bus names are often chosen for convenience to be the same string. Remember that they serve distinct purposes: the interface name identifies a contract between the client and the service that is implemented by a bus object living in a bus attachment; and the well-known name identifies the service in a consistent way to clients wishing to connect to that attachment.

To use a well-known name, an application (by way of a bus attachment) must make a request to the bus router to use that name. If the well-known name is not already in use by another application, exclusive use of the well-known name is granted. This is how, at any time, well-known names are guaranteed to represent unique addresses on the bus.

Typically, a well-known name implies a contract that the associated bus attachment implements a collection of bus objects and therefore some concept of a usable service. Since bus names provide a unique address on the distributed bus, they must be unique across the bus. For example, one could use the bus name, `org.alljoyn.sample.chat`, which would indicate that a bus attachment of the same name would be implementing a chat service. By virtue of the fact that it has taken that name, one could infer that it implements a corresponding `org.alljoyn.sample.chat` interface in a bus object located at object path `/org/alljoyn/sample/chat`.

The problem with this is that in order to "chat", one would expect to see another similar component on the AllJoyn bus indicating that it also supports the chat service. Since bus names must uniquely identify a bus attachment, there is a requirement to append some form of suffix to ensure uniqueness. This could take the form of a user name, or a unique number. In the chat example, one could then imagine multiple bus attachments:

```
org.alljoyn.sample.chat.bob

org.alljoyn.sample.chat.carol
```

In this case, the well-known name prefix `org.alljoyn.sample.chat.` acts as the service name, from which one can infer the existence of the chat interface and object implementations. The suffixes, `bob` and `carol`, serve to make the instance of the well-known name unique.

This leads to the question of how services are located in the distributed system. The answer is via service advertisement and discovery by clients.

Advertisements and discovery

There are two facets to the problem of service advertisement and discovery. As described above, even if the service resides on the local segment of the AllJoyn bus, one needs to be able to see and examine the well-known names of all of the bus attachments on the bus in order to determine that one of them has a specific service of interest. A more interesting problem occurs when one considers how to discover services that are not part of an existing bus segment.

Consider what might happen when one brings a device running the AllJoyn framework into the proximity of another. Since the two devices have been physically separated, there is no way for the two involved bus routers to have any knowledge of the other. How do the routing nodes determine that the other exists, and how do they determine that there is any need to connect to each other and form a logical distributed AllJoyn bus?

The answer is through the AllJoyn service advertisement and discovery facility. When a service is started on a local device, it reserves a given well-known name and then advertises its existence to other devices in its proximity. The AllJoyn framework provides an abstraction layer that makes it possible for a service to do an advertise operation that may be communicated transparently via underlying technologies, such as Wi-Fi, Wi-Fi Direct, or other/future wireless transports. Neither the client nor the service require any knowledge of how these advertisements are managed by the underlying technology.

For example, in a contacts-exchanging application, one instance of the application may reserve the well-known name `org.alljoyn.sample.contacts.bob` and advertise the name. This might result in one or more of the following: a UDP multicast over a connected Wi-Fi access point, a pre-association service advertisement in Wi-Fi Direct, or a Bluetooth Service Discovery Protocol message. The mechanics of how the advertisement is communicated do not necessarily concern the advertiser. Since a contacts-exchange application is conceptually a peer-to-peer application, one would expect the second phone to also advertise a similar service, for example, `org.alljoyn.sample.contacts.carol`. Client applications may declare their interest in receiving advertisements by initiating a discovery operation. For example, it may ask to discover instances of the contacts service

both uses the same service

as specified by the prefix `org.alljoyn.sample.contacts`. In this case, both devices would make that request.

As soon as the phones enter the proximity of the other, the underlying AllJoyn systems transmit and receive the advertisements over the available transports. Each will automatically receive an indication that the corresponding service is available.

Since a service advertisement can receive over multiple transports, and in some cases it requires additional low-level work to bring up an underlying communication mechanism, there is another conceptual part to the use of discovered services. This is the communication session.

Sessions

The concepts of bus names, object paths, and interface names have been previously discussed. Recall that when an entity connects to an AllJoyn bus, it is assigned a unique name. Connections (bus attachments) may request that they be granted a well-known name. The well-known name is used by clients to locate or discover services on the bus. For example, a service may connect to an AllJoyn bus and be assigned the unique name `:1.1` by the bus. If a service wants other entities on the bus to be able to find it, the service must request a well-known name from the bus, for example, `com.companyA.ProductA` (remember that a unique instance qualifier is usually appended).
This name implies at least one bus object that implements some well-known interface for it to be meaningful. Usually, the bus object is identified within the connection instance by a path with the same components as the well-known name (this is not a requirement, it is only a convention). In the example, the path to the bus object corresponding to the bus name `com.companyA.ProductA` might be `/com/companyA/ProductA`.

In order to understand how a communication session from a client bus attachment to a similar service attachment is formed and to provide an end-to-end example, it is useful to compare and contrast the AllJoyn mechanism to a more familiar mechanism.

Postal address analogy

In the AllJoyn framework, a service requests a human-readable name so it can advertise itself with a well-known and well-understood label. Well-known names must be translated into unique names for the underlying network to properly route information, for example:

```
Well-known-name:org.alljoyn.sample.chat

Unique name::1.1
```

This tells us that the well-known name advertised as `org.alljoyn.sample.chat` corresponds to a bus attachment that has been assigned the unique name `:1.1`. One can think of this in the same way as a business has a name and a postal address. To continue the analogy, a common situation arises when a business is located in a building along with other businesses. In such a situation, one might find a business address further qualified by a suite number. Since AllJoyn bus attachments are capable of providing more than one service, there must also be a way to identify more than one destination on a particular attachment. A "contact port number" corresponds to the suite number destination in the postal address analogy.

Just as one may send a letter by the national mail system (U.S. Post Office, La Poste Suisse) or a private company (Federal Express, United Parcel Service) and by different urgencies (overnight, two-day, overland delivery), when contacting a service using the AllJoyn framework, one must specify certain desired characteristics of the network connection to provide a complete delivery specification (e.g., reliably delivered messages, reliably delivered unstructured data, or unreliably delivered unstructured data).

Notice the separation of the address information and the delivery information in the example above. Just as one can contemplate choosing several ways to get a letter from one place to another, it will become evident that one can choose from several ways to get data delivered using the AllJoyn system.

The AllJoyn session

Just as a properly labeled postal letter has "from" and "to" addresses, an AllJoyn session requires equivalent "from" and "to" information. In the case of an AllJoyn system, the from address would correspond to the location of the client component and the to address would relate to the service.

Technically, these from or to addresses, in the context of computer networking, are called half-associations. In the AllJoyn framework, this to (service) address has the following form:

```
{session options, bus name, session port}
```

The first field, session options, relates to how the data is moved from one side of the connection to the other. In an IP network, choices might be TCP or UDP. In the AllJoyn framework, these details are abstracted and so choices might be, "message-based", "unstructured data", or "unreliable unstructured data". A service destination is specified by the well-known name the corresponding bus attachment has requested.

Similar to the suite number in the postal example, the AllJoyn model has the concept of a point of delivery "inside" the bus attachment. In the AllJoyn framework, this is called a session port. Just as a suite number has meaning only within a given building, the session port has meaning only within the scope of a given bus attachment. The existence and values of contact ports are inferred from the bus name in the same way that underlying collections of objects and interfaces are inferred.

The from address, corresponding to the client information, is similarly formed. A client must have its own half-association in order to communicate with the service.

```
{session options, unique name, session ID}
```

It is not required for clients to request a well-known bus name, so they provide their unique name (such as `:1.1`). Since clients do not act as the destination of a session, they do not provide a session port, but are assigned a session ID when the connection is established. Also during the session establishment procedure, a session ID is returned to the service. For those familiar with TCP networking, this is equivalent to the connection establishment procedure used in TCP, where the service is contacted over a well-known port. When the connection is established, the client uses an ephemeral port to describe a similar half-association.

During the session establishment procedure, the two half-associations are effectively joined:

```
{session options, bus name, session port}    Service

{session options, unique name, session ID}    Client
```

Notice that there are two instances of the session options. When communication establishment begins, these may be viewed as supported session options provided by the service and requested session options provided by the client. Part of the session establishment procedure consists of negotiating an actual final set of options to be used in

the session. Once a session has been formed, the half-associations of the client and service side describe a unique AllJoyn communication path:

```
{session options, bus name, unique name, session ID}
```

During the session establishment procedure, a logical networking connection is formed between the communicating routing nodes. This may result in the creation of a wireless radio topology management operation. If such a connection already exists, it is re-used. A newly created underlying router-to-router connection is used to perform initial security checks, and once this is complete, the two routers have effectively joined the two separate AllJoyn software bus segments into the larger virtual bus.

Because issues regarding end-to-end flow control of the underlying connection must be balanced with topological concerns in some technologies, the actual connection between the two communicating endpoints (the "from" client and the "to" service) may or may not result in a separate communication channel being formed. In some cases it is better to flow messages over an ad hoc topology and in some cases it may be better to flow messages directly over a new connection (TCP/IP). This is another of the situations that may require deep understanding of the underlying technology to resolve, and which the AllJoyn framework happily accomplishes for you. A user need only be aware that messages are routed correctly over a transport mechanism that meets the abstract needs of the application.

Self-join feature

In AllJoyn releases up to R14.06, it was impossible for applications to join a session they themselves hosted. For applications that consume information or services they themselves also provide, this created an asymmetry: they had to treat the bus objects they hosted themselves differently from those hosted by other peers. The self-join feature removes this asymmetry by allowing applications to join the sessions they themselves host. Consequently, a locally hosted bus object can be treated in exactly the same way as a remotely hosted bus object.

Determining the presence of a peer - pinging and auto-pinging

Sometimes, a application needs to know which peers are present on the communication channel ("the wire") and which aren't. For this reason, a PING API was introduced in

version 14.06. This PING API allows to determine whether a peer is up or not. However, for this API, the responsibility for using the PING API was with the Application, which periodically needed to ping the peers. From Release 14.12 onwards, an automatic PING or Auto-Pinger is introduced. This Auto-Pinger performs the periodic peer detection, relieving the application of having to do it.

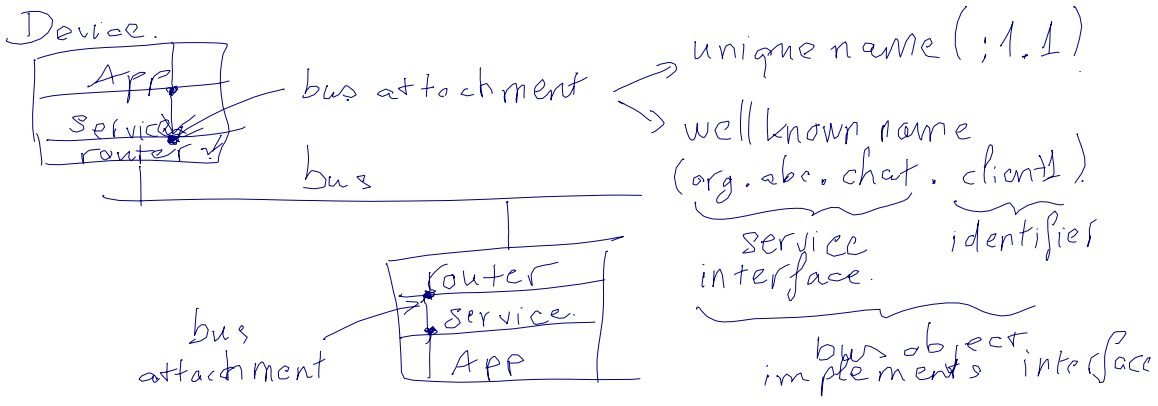
Bringing it all together

The AllJoyn framework aims to provide a software bus that manages the implementation of advertising and discovering services, providing a secure environment, and enabling location-transparent remote method invocation. A traditional client/service arrangement is enabled, and peer-to-peer communications follow by combining the aspects of client and services.

The most basic abstraction in the AllJoyn framework is the software bus that ties everything together. The virtual distributed bus is implemented by AllJoyn routing nodes which are background programs running on each device. Clients and services (and peers) connect to the bus via bus attachments. The bus attachments live in the local processes of the clients and services and provide the interprocess communication that is required to talk to the local AllJoyn router.

Each bus attachment is assigned a unique name by the system when it connects. A bus attachment can request to be granted a unique human-readable bus name that it can use to advertise itself to the rest of the AllJoyn world. This well-known bus name lives in a namespace that looks like a reversed domain name and encourages self-management of the namespace. The existence of a bus attachment of a specific name implies the further existence of at least one bus object that implements at least one interface specified by a name. Interface names are assigned out of a namespace that is similar, but has a different meaning than bus names. Each bus object lives in a tree structure rooted at the bus attachment and described by an object path that looks like a Unix filesystem path.

The following figure shows a hypothetical arrangement of how all of these pieces are related.



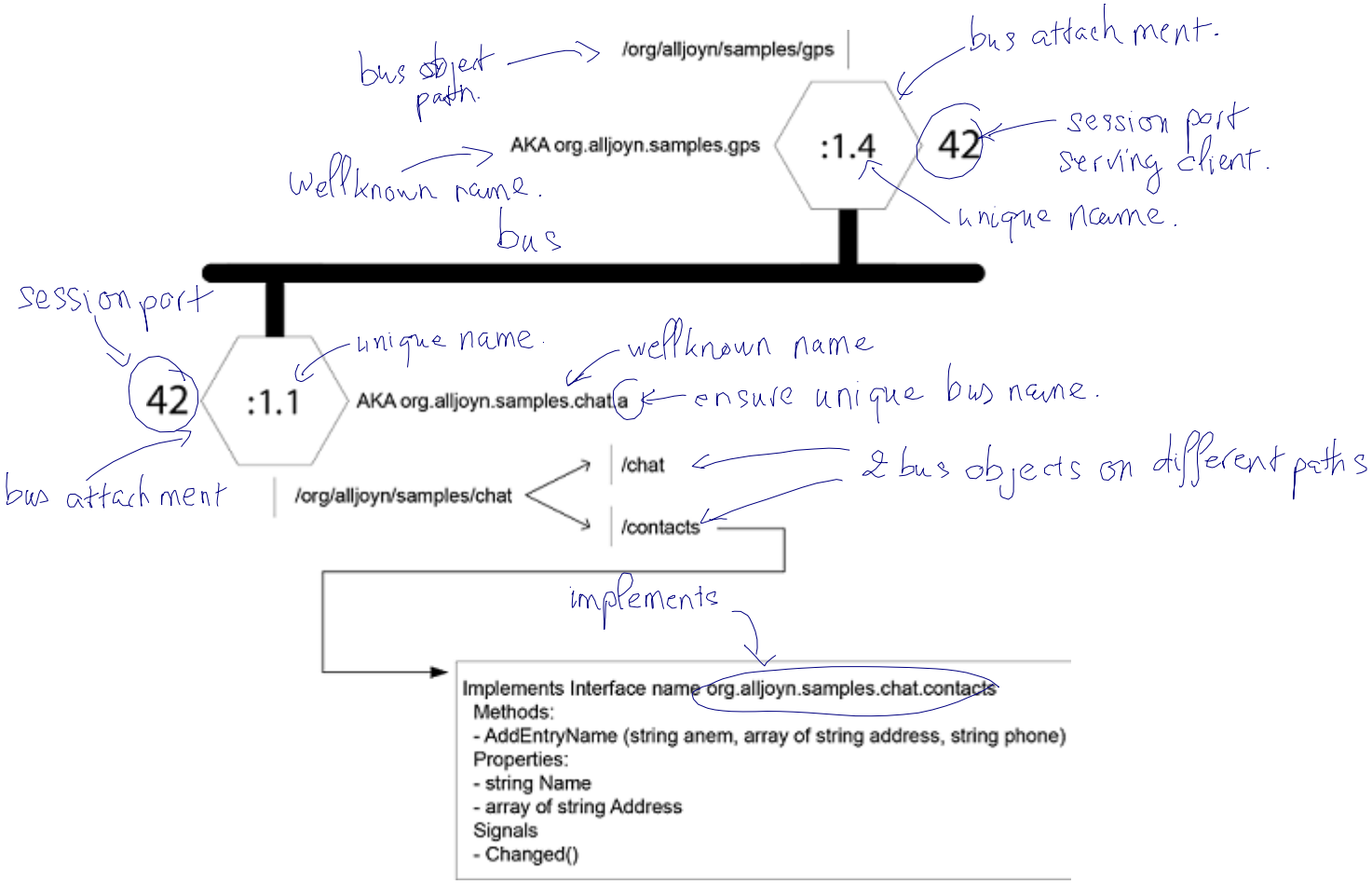


Figure: Overview of a hypothetical AllJoyn bus instance

At the center is the dark line representing the AllJoyn bus. The bus has "exits" which are the BusAttachments assigned the unique names `:1.1` and `:1.4`. In the figure, the BusAttachment with the unique name of `:1.1` has requested to be known as `org.alljoyn.samples.chat.a` and has been assigned the corresponding well-known bus name. The "a" has been added to ensure that the bus name is unique.
There are a number of things implied by taking on that bus name. First, there is a tree structure of bus objects that resides at different paths. In this hypothetical example, there are two bus objects. One is at the path `/org/alljoyn/samples/chat/chat` and which presumably implements an interface suitable for chatting. The other bus object lives at the path `/org/alljoyn/samples/chat/contacts` and implements an interface named `org.alljoyn.samples.chat.contacts`. Since the given bus object implements the interface, it must provide implementations of the corresponding bus methods, bus signals, and bus properties.

The number 42 represents a contact session port that clients must use to initiate a communication session with the service. Note that the session port is unique only within the context of a particular bus attachment, so the other bus attachment in the figure may also use 42 as its contact port as shown.

After requesting and being granted the well-known bus name, a service will typically advertise the name to allow clients to discover its service. The following figure shows a service making an advertise request to its local router. The router, based on input from the service, decides what network medium-specific mechanism it should use to advertise the service and begins doing so.

Figure: Service performs an Advertise

When a prospective client wants to locate a service for consumption, it issues a find name request. Its local router device, again based on input from the client, determines the best way to look for advertisements and probes for advertisements.

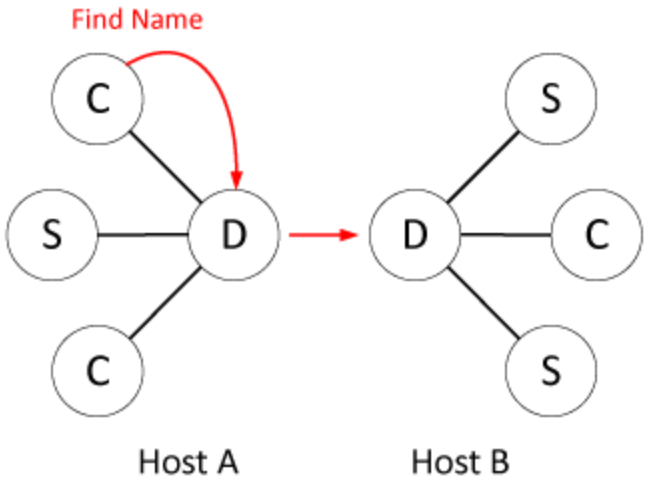


Figure: Client requests to Find Name

Once the devices move into proximity, they begin hearing each other's advertisements and discovery requests over whichever media are enabled. The following figure shows how the router hosting the service hears the discovery requests and responds.

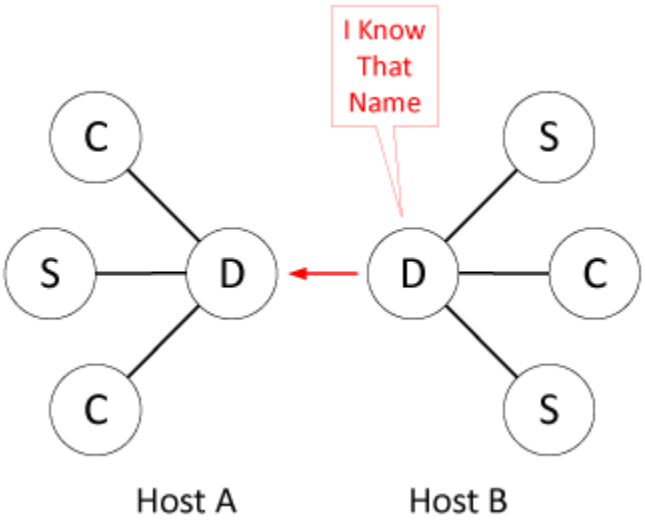


Figure: Router reports Found Name

Finally, the following figure shows the client receiving an indication that there is a new router in the area that is hosting the desired service.

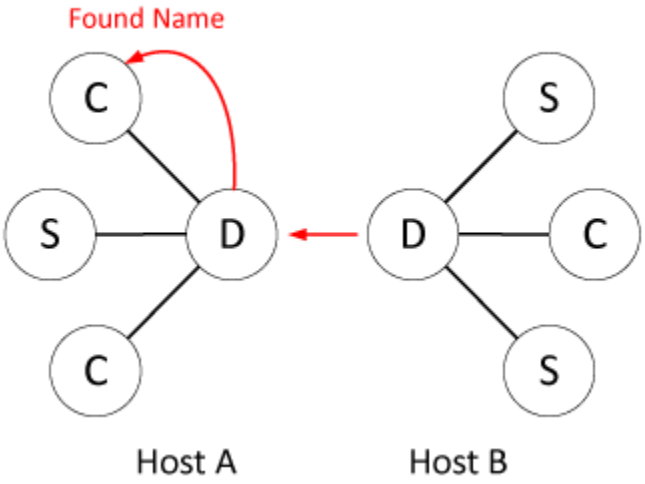


Figure: Client discovers service

The client and service sides of the developing scenario both use methods and callbacks on their bus attachment object to make the requests to orchestrate the advertisement and discovery process. The service side implements bus objects to provide its service, and the client will expect to use a proxy object to provide an easy-to-use interface for communicating with the service. This proxy object will use an AllJoyn ProxyBusObject to

orchestrate communication with the service and provide for the marshaling and unmarshaling of method parameters and return values.

Before remote methods can be called, a communication session must be formed to effectively join the separate bus segments. Advertisement and discovery are different from session establishment. One can receive an advertisement and take no action. It is only when an advertisement is received, and a client decides to take action to join a communication session, that the buses are logically joined into one. To accomplish this, a service must create a communication session endpoint and advertise its existence; and a client must receive that advertisement and request to join the implied session. The service must define a half-association before it advertises its service. Abstractly this will look something like the following:

```
{reliable IP messages, org.alljoyn.samples.chat.a, 42}
```

This indicates that it will talk to clients over a reliable message-based transport, has taken the well-known bus name indicated, and expects to be contacted at session port 42. This is the situation seen in the hypothetical bus instance figure.

Assume that there is a bus attachment with the unique name `:2.1` wanting to connect from a physically remote routing node. It will provide its half association to the system and a new session ID will be assigned and communicated to both sides of the conversation:

```
{reliable IP messages, org.alljoyn.samples.chat.a, :2.1, 1025}
```

The new communication session will use a reliable messaging protocol implemented using the IP protocol stack which will exist between the bus attachment named `org.alljoyn.samples.chat.a` (the service) and the bus attachment named :2.1 (the client). The session ID used to describe the session is assigned by the system and is 1025 in this case.

As a result of establishing the end-to-end communication session, the AllJoyn system takes whatever actions are appropriate to create the virtual software bus shown in the distributed bus figure. Note that this is a virtual picture, and what may have actually happened is that a Wi-Fi Direct peer-to-peer connection was formed to host a TCP connection, or a Wireless access point was used to host a UDP connection, depending on the provided session options. Neither the client nor the service is aware that this possibly very difficult job was completed for them.

At this point, authentication can be attempted if desired and then the client and service begin communicating using the RMI model.

Of course, the scenario is not limited to one client on one device and one service on another device. There may be any number of clients and any number of services (up to a limit of device or network capacity) combining to accomplish some form of cooperative work. Bus attachments may take on both client and service personalities and implement peer-to-peer services. AllJoyn routers take on the hard work of forming a manageable logical unit out of many disparate components and routing messages. Additionally, the nature of the interface description and language bindings allow interoperability between components written in different programming languages.

High-Level System Architecture

From the perspective of a user of the AllJoyn system, the most important piece of the architecture to understand is that of a client, service, or peer. From a system perspective, there is really no difference between the three basic use cases; there are simply different usage patterns of the same system-provided functionality.

Clients, services, and peers

The following figure shows the architecture of the system from a user (not AllJoyn router) perspective.

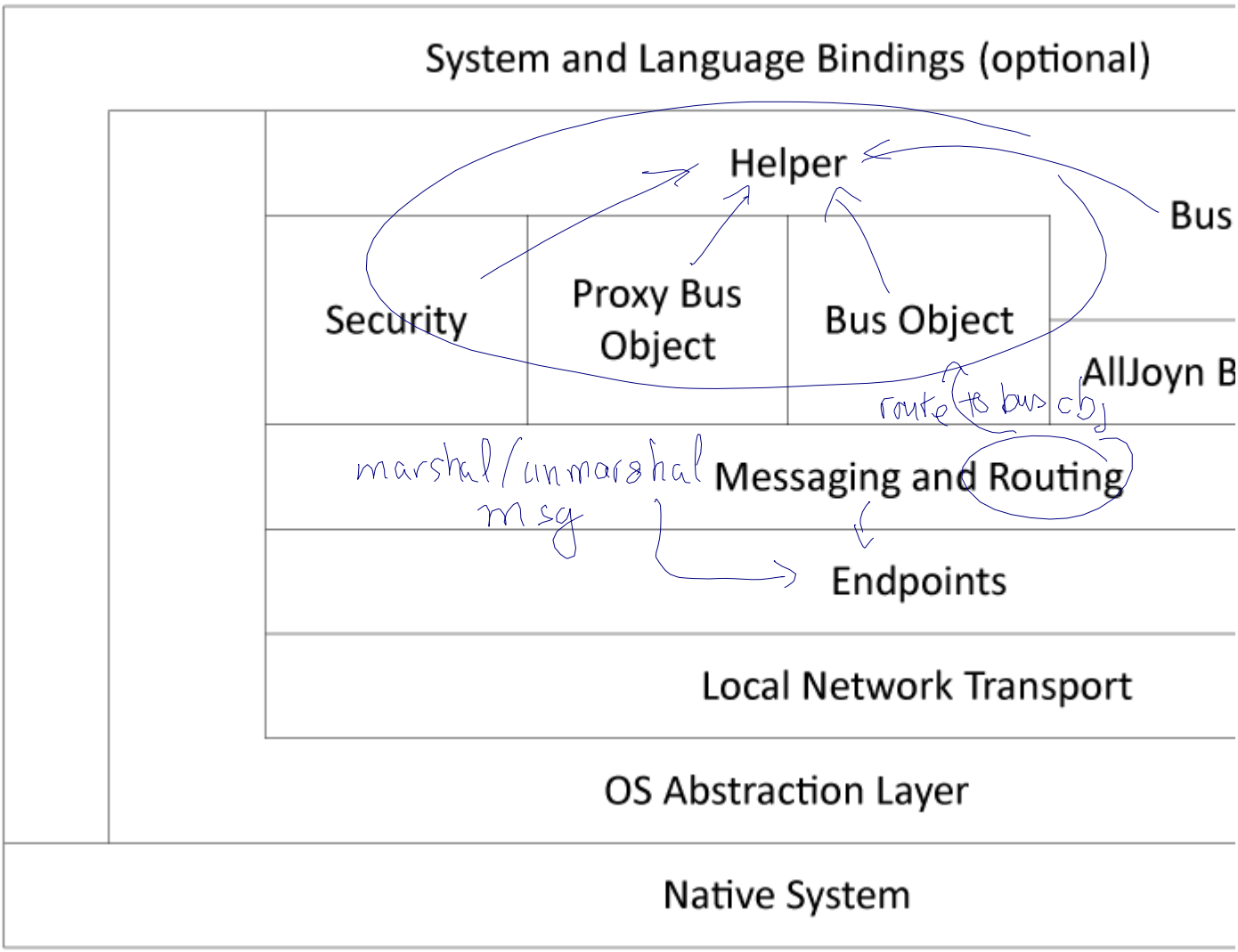


Figure: Basic client, service, or peer architecture

At the highest level are the language bindings. The AllJoyn system is written in C++, so for users of this language, no bindings are required. However, for users of other languages, such as Java or JavaScript, a relatively thin translation layer called a language binding is provided. In some cases, the binding may be extended to offer system-specific support. For example, a generic Java binding will allow the AllJoyn system to be used from a generic Java system that may be running under Windows or Linux; however, an Android system binding may also be provided which more closely integrates the AllJoyn system into Android-specific constructs such as a service component in the Android application framework.

The system and language bindings are built on a layer of helper objects which are designed to make common operations in the AllJoyn system easier. It is possible to use much of the AllJoyn system without using these helpers; however, their use is encouraged since it provides another level of abstract interface. The bus attachment, mentioned in the previous chapters, is a critical helper without which the system is unusable. In addition to the several critical functions provided, a bus attachment also provides convenience functions to make management of and interaction with the underlying software bus much easier.

Under the helper layer is the messaging and routing layer. This is the home of the functionality that marshals and unmarshals parameters and return values into messages that are sent across the bus. The routing layer arranges for the delivery of inbound messages to the appropriate bus objects and proxies, and arranges for messages destined for other bus attachments to be sent to an AllJoyn router for delivery.

The messaging and routing layer talks to an endpoint layer. In the lower levels of the AllJoyn system, data is moved from one endpoint to another. This is an abstract communication endpoint from the perspective of the networking code. Networking abstractions are fully complete at the top of the endpoint's layer, where there is essentially no difference between a connection over a non Wi-Fi radio (Bluetooth) and a connection over a wired Ethernet.

Endpoints are specializations of transport mechanism-specific entities called transports, which provide basic networking functionality. In the case of a client, service, or peer, the only network transport used is the local transport. This is a local interprocess communication link to the local AllJoyn bus router. In Linux-based systems, this is a Unix-domain socket connection, and in Windows-based systems this is a TCP connection to the local router.

The AllJoyn framework provides an OS abstraction layer to provide a platform on which the rest of the system is built, and at the lowest level is the native system.

Routers

AllJoyn routers are the glue that holds the AllJoyn system together. As previously discussed, routers are programs that run in the background, waiting for interesting events to happen and responding to them. Because these events are usually external, it is better to approach the router architecture from a bottom-up perspective.

At the lowest level of the router architecture figure below, resides the native system. We use the same OS abstraction layer as we do in the client architecture to provide common abstractions for routers running on Linux, Windows, and Android. Running on the OS abstraction layer, we have the various low-level networking components of the router. Recall that clients, services, and peers only use a local interprocess communication mechanism to talk to a router, so it is the router that must deal with the various available transport mechanisms on a given platform. Note the "Local" transport in the router architecture figure which is the sole connection to the AllJoyn clients, services, and peers running on a particular host.

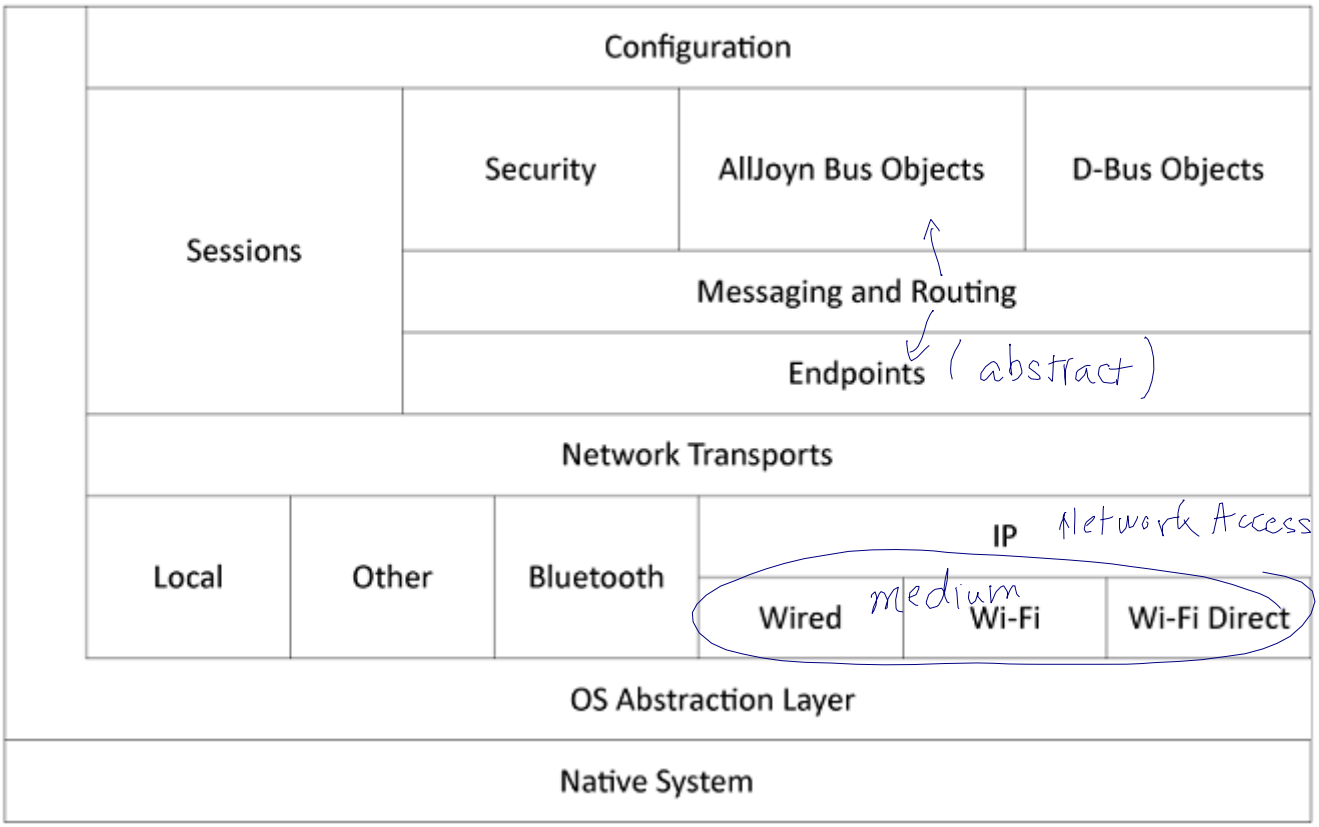


Figure: Basic router architecture

For example, a Bluetooth transport would handle the complexities of creating and managing piconets in the Bluetooth system. Additionally, a Bluetooth transport provides service advertisement and discovery functions appropriate to Bluetooth, as well as providing reliable communications. Bluetooth and other transports would be added at this transport layer along side the IP transport.

The wired, Wi-Fi, and Wi-Fi Direct transports are grouped under an IP umbrella since all of these transports use the underlying TCP-IP network stack. There are sometimes significant differences regarding how service advertisement and discovery is accomplished, since this functionality is outside the scope of the TCP-IP standard; so there are modules dedicated to this functionality.

The various technology-specific transport implementations are collected into a Network Transports abstraction. The Sessions module handles the establishment and maintenance of communication connections to make a collection of routers and AllJoyn applications appear as a unified software bus.

AllJoyn routers use the endpoint concept to provide connections to local clients, services, and peers but extend the use of these objects to bus-to-bus connections which are the transports used by routers to send messages from host-to-host.

In addition to the routing functions implied by these connections, an AllJoyn router provides its own endpoints corresponding to bus objects used for managing or controlling the software bus segment implemented by the router. For example, when a service requests to advertise a well-known bus name, what actually happens is that the helper on the service translates this request into a remote method call that is directed to a bus object implemented on the router. Just as in the case of a service, the router has a number of bus objects living at associated object paths which implement specific named interfaces. The low-level mechanism for controlling an AllJoyn bus is sending remote method invocations to these router bus objects.

The overall operation of certain aspects of router operation are controlled by a configuration subsystem. This allows a system administrator to specify certain permissions for the system and provides the ability to arrange for on-demand creation of services. Additionally, resource consumption may be limited by configuration of the router, allowing a system administrator to, for example, limit the number of TCP connections active at any given time. There are options which allow system administrators to mitigate the effects of certain denial-of-service attacks, by limiting the number of connections which are currently authenticating, for example.

Summary

The AllJoyn framework is a comprehensive system designed to provide a framework for deploying distributed applications on heterogeneous systems with mobile elements.

The AllJoyn framework provides solutions, building on proven technologies and standard security systems, that address the interaction of various network technologies in a coherent, systematic way. This allows application developers to focus on the content of their applications without requiring a large amount of low-level networking experience.

The AllJoyn system is designed to work together as a whole and does not suffer from inherent impedance mismatches that might be seen in ad-hoc systems built from various pieces. We believe that the AllJoyn system can make development and deployment of distributed applications significantly simpler than those developed on other platforms.

ALLJOYN® THIN CORE

Overview

AllJoyn is an open-source software system that provides an environment for distributed applications running across different device classes, with an emphasis on mobility, security, and dynamic configuration. AllJoyn is "platform-neutral", meaning it was designed to be as independent as possible of the specifics of the operating system, hardware, and software of the device on which it is running.

Components of the AllJoyn Standard Core Library (AJSCL) are designed to run on Microsoft Windows, Linux, Android, iOS, OS X, OpenWRT, and the Unity plug-in for internet browsers. A common characteristic of all of these software systems is that they run on general-purpose computers. General purpose computers usually have significant amounts of memory, available energy, and computing power, along with significant operating systems that support multiple processes and multiple threads with multiple standard language environments.

An embedded system, on the other hand, is one designed to provide specific functionality running on a microcontroller embedded within a larger device. Since an embedded system need only perform a specific function or a small number of functions, engineers are free to optimize them 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. AllJoyn Thin Core Library (AJTCL) is designed to bring the benefits of the AllJoyn distributed programming environment to embedded systems.

Since the operating environment in which an AJTCL will run may be very constrained, an AllJoyn component running on such systems must live within those constraints. This means, specifically, that we do not have the luxury of bundling in an AllJoyn router (which requires multi-threading), having many network connections, and using relatively large amounts of RAM and ROM. We do not have the luxury of running an object-oriented programming environment that includes alternate language bindings. Because of this, the AJTCL consists only of what amounts to a bus attachment (see the **Introduction to the AllJoyn Framework**) written solely in the C language. The data structures corresponding to interfaces, methods, signals, properties, and bus objects are highly optimized for space, and the developer APIs are, therefore, quite different.

Although the APIs may be different, all of the major conceptual blocks found in AJSCL can be found in AJTCL systems; they just take on a more compact form or are actually run remotely on another, more capable machine.

NOTE: When we mention the AllJoyn Standard Library (AJSCL), we explicitly refer to the versions of these components that run on general purpose computers

Conceptual Model

As implied in the previous section, most high-level abstractions used in AJTCL are identical to those in the AJSCL system. The **Introduction to the AllJoyn Framework** has a section titled Conceptual Overview that walks you through these abstractions. In the Conceptual Overview section, we assume that the reader is familiar with the abstractions introduced in that document, so we will only touch on the differences that are required to understand the AJTCL architecture.

AllJoyn Thin Core Library is still AllJoyn

It is important to understand that AJTCL is part of the AllJoyn framework. A Thin Core Library is completely interoperable with AJSCL. Since the AllJoyn network wire protocol is completely implemented on both types of such a system, AJSCL can be completely unaware of the fact that they are talking to Thin Core Libraries, and vice versa.

Recall from the **Introduction to the AllJoyn Framework** that the basic structure of an AllJoyn distributed bus consists of multiple bus segments residing on physically separate host computers.

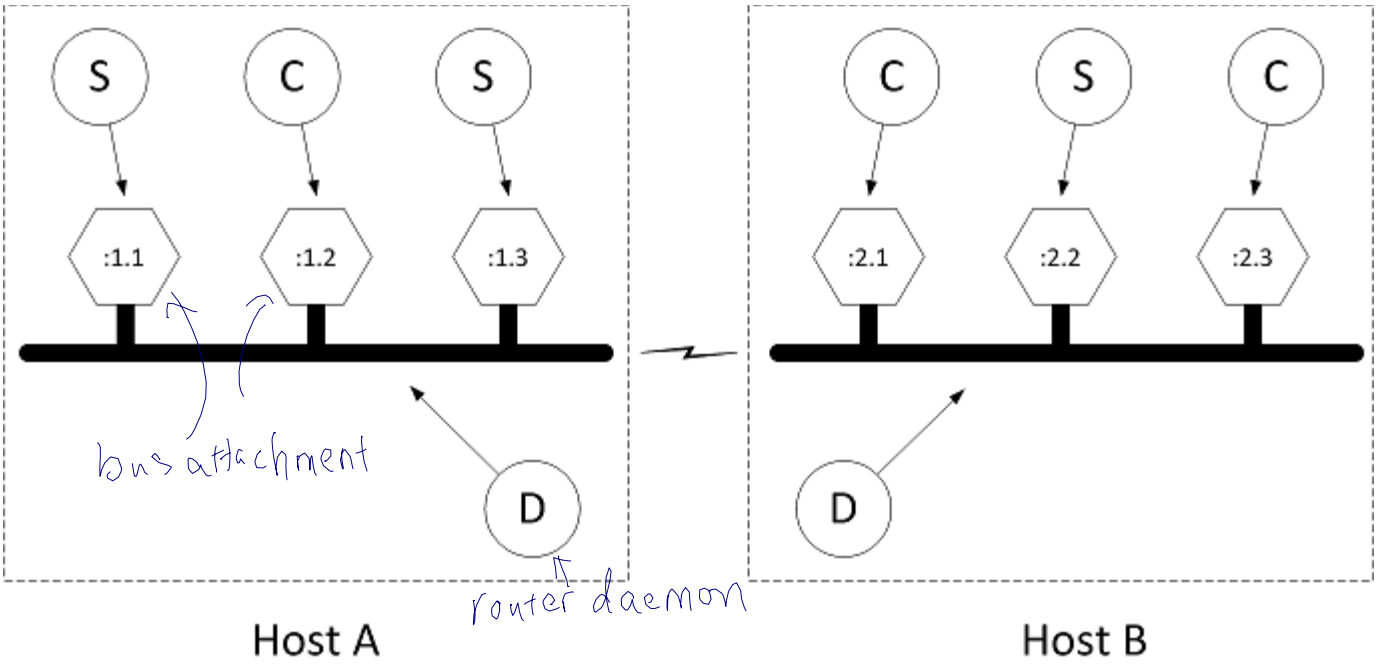


Figure: AllJoyn distributed bus

Recall that each bus segment is located on a given host computer, as illustrated by the dotted squares labeled Host A and Host B in the figure. Each bus segment is implemented by an AllJoyn router (shown as the bubbles labeled D in the figure). There may be several bus attachments on a host, each connected to the local daemon (illustrated by hexagons). These hexagons are refined to be services (S) or clients (C).

Since the host computer running AJTCL typically does not have the resources to run a router, the AllJoyn architecture changes things such that to connect to the distributed bus the Thin Core Library borrows an AllJoyn router running on another host computer.

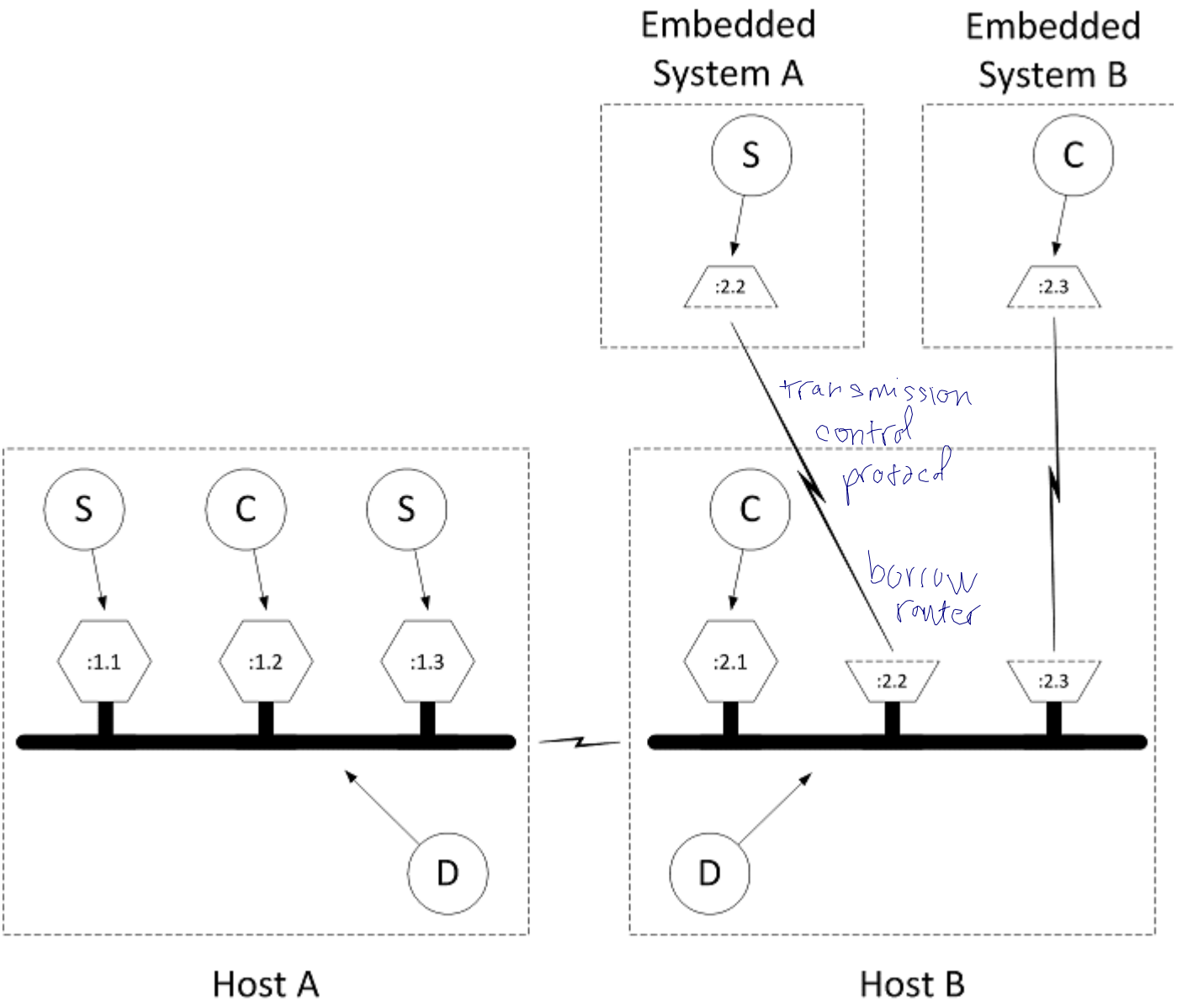


Figure: AllJoyn distributed bus with thin core libraries

Notice that Embedded System A and Embedded System B are not the same devices as Host B, which is running the router that manages the distributed bus segment on which the embedded devices reside. The connection between the embedded systems running AJTCL and the router hosting the bus segment is made through Transmission Control Protocol (TCP).

The network traffic flowing between the embedded systems and the routers are AllJoyn messages implementing bus methods, bus signals, and properties flowing over their respective sessions, as described in **Introduction to the AllJoyn Framework**.

It is sometimes desirable to allow AJTCL devices to connect to and borrow any old router found in the proximity. We call these untrusted relationships (from the router perspective). It is also sometimes desirable to allow only particular AJTCL devices to connect to specific routers. We call these trusted relationships (again, from the router perspective).

These relationships are established using a discovery and connection process that is conceptually similar to the discovery and connection process of clients and services. An AllJoyn router conveys its willingness to host a given collection of AJTCL devices by advertising a well-known name. This advertisement may be driven either by router configuration or by an advertisement specifically made by an AllJoyn component. When a connection attempt is made to any router as a result of a discovery event, a router expecting trusted relationships may choose to challenge a particular Thin Core Library (or impersonator of a Thin Core Library) to produce a credential. In the case of an untrusted relationship, the router may choose to simply allow any connection attempt. In the case of an untrusted connection, the involved router will not allow the Thin Library to perform any operations that will cause sessions to be established with components off the local device (and which, therefore, correspond to a "service that costs you money").

As implied above, the connection process for an AJTL device is split into three phases:

- Discovery phase
- Connection phase
- Authentication phase
  The discovery phase works just like service advertisement and discovery as described in **Introduction to the AllJoyn Framework**, with two exceptions. The first exception is that advertisements for the purpose of AJTL discovery are typically "quiet" advertisements. This simply means that the advertisements are not sent gratuitously by the router.

  The second exception is that responses to quiet advertisements are sent quietly - we call these quiet responses. This means that the responses are unicast back to the requester instead of being multicast as they are in "active" advertisements. The primarily reason for this change is to allow embedded devices that do not fully implement multicast reception to participate in AllJoyn distributed systems.

What is an AllJoyn Thin Core Library device?

One typically thinks of an AJTCL device as conceptually similar to a Sensor Node (SN) in a Wireless Sensor Network (WSN). Sensor nodes are typically sensors/actuators that are small in size and constrained in energy, computing power, memory, or other resources. They are able to sense their surroundings, communicate events to the outside world, and possibly take actions based on internal processing or as a result of external events. This is a very broad definition, and a small sampling of the sort of devices that might fit into such a definition could be:

- Light switches
- Thermostats
- Air conditioners
- Vent dampers
- Smoke detectors
- Motion detectors
- Humidity detectors
- Microphones
- Speakers
- Earphones
- Doors
- Doorbells
- Ovens
- Refrigerators
- Toasters

There is a large amount of literature available that discusses wireless sensor networks (WSNs). AllJoyn systems are distinguished from such networks in that WSNs typically use

self-organizing multi-hop ad hoc wireless networks where security is not a major concern; whereas the AllJoyn framework will most likely run on infrastructure-mode Wi-Fi networks to which a given device must be associated and authenticated. In order to accomplish the secure admission to a Wi-Fi network, AJTCL uses a process called "onboarding". The Onboarding service framework allows a Thin Core Library device, which presumably has no friendly user interface, to learn enough information about its destination network to accomplish the admission and authentication processes required to join that network. The Onboarding service framework is defined in detail in a dedicated document.

In its role as a kind of sensor node, an AJTCL device typically implements a service in the AllJoyn sense. It senses its surroundings using attached hardware and communicates events to the outside world through AllJoyn signals. It can take actions as a result of external events, either by listening for signals from other devices or by responding to Remote Method Invocations from AllJoyn clients, as discussed in **Introduction to the AllJoyn Framework**.

Thin Core Library Architecture

Since the AllJoyn Thin Core Library (AJTCL) must run in devices that are constrained in energy, processing power, and memory, such devices do not have the luxury of using the same architecture as a general-purpose computer system running AllJoyn Standard Core Library (AJSCL).

The layered architecture of an AJSCL or service process is reproduced below.

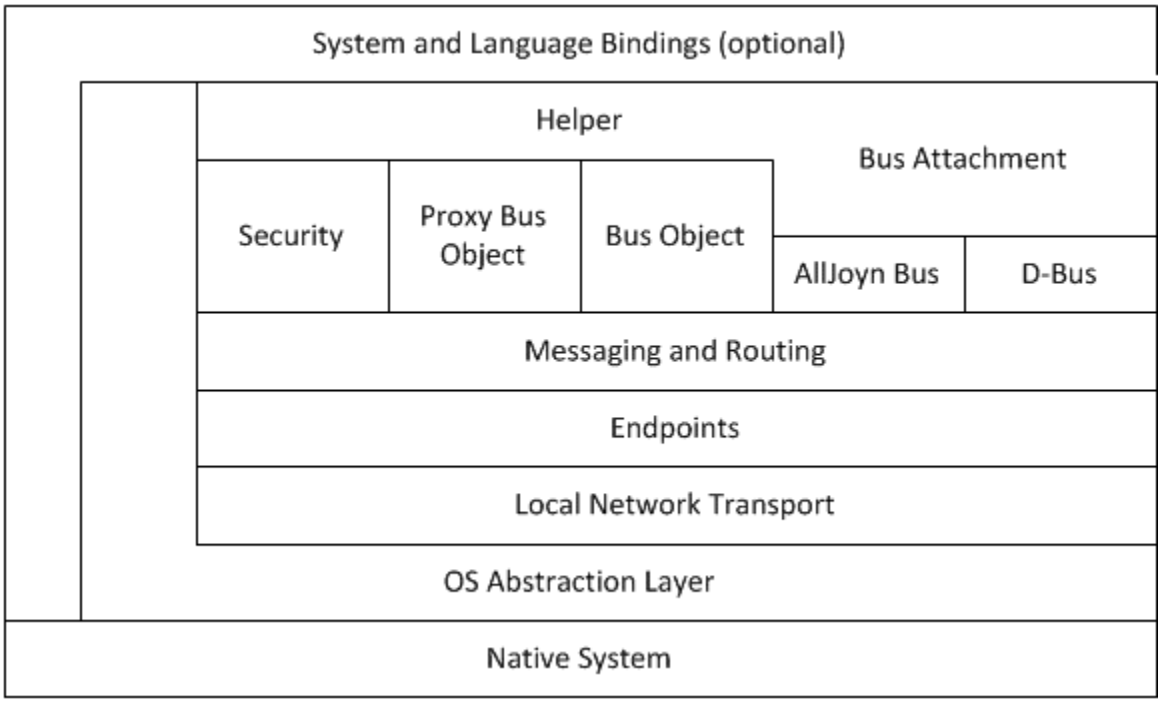
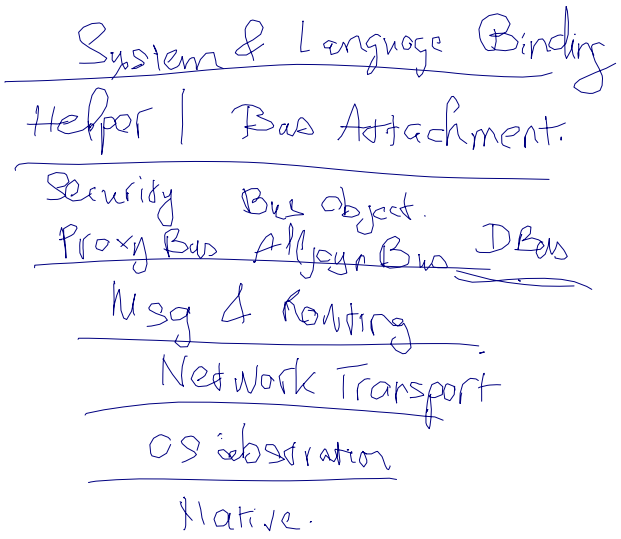


Figure: AJSCL layering

See the **Introduction to the AllJoyn Framework** for a more detailed discussion of these layers.

The important observation to make at this point is that each AllJoyn client or service reproduces this layering in every process representing an AllJoyn application.

Every AJSCL-enabled host needs to have at least one AllJoyn router. This router may reside in its own process in the standalone router case, or it may be co-located with an application in the bundled router case. The layered architecture of an AJSCL router is reproduced below.



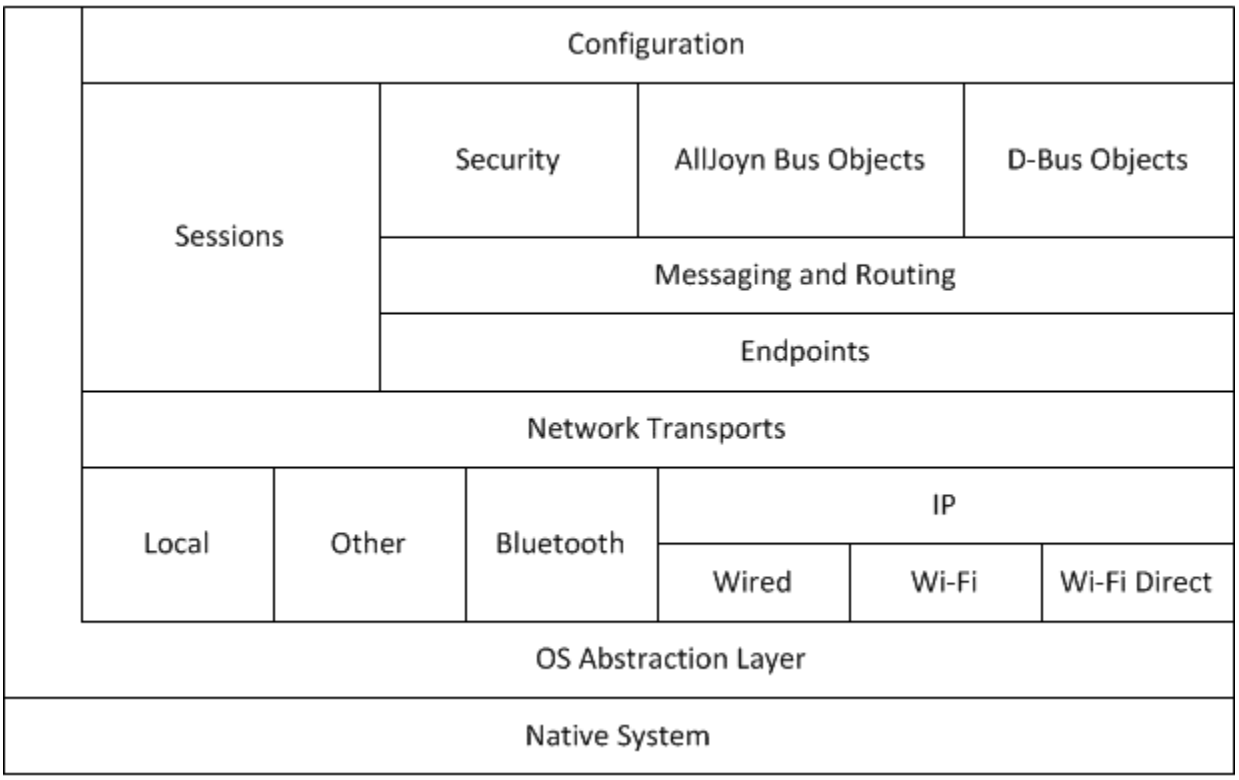
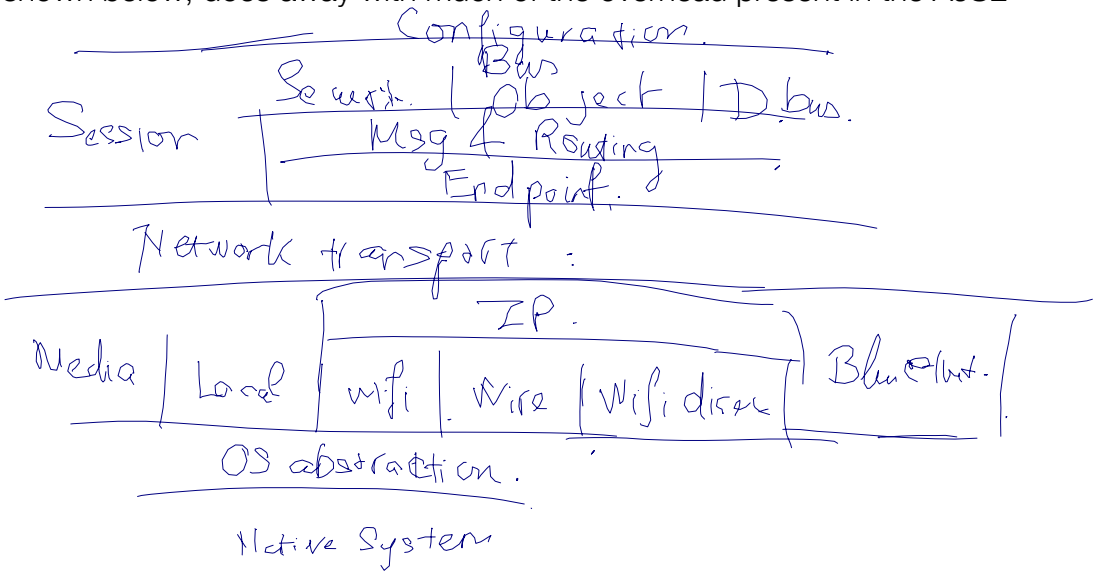


Figure: AJSCL router layering

Notice that the router adds additional support for routing messages between router, along with the capacity to use a multiple network transport mechanisms such as Wi-Fi Direct. This is a significant amount of functionality and comes at a considerable cost in computing power, energy, and memory.

Clearly, it is not possible to run this significant amount of code in a constrained embedded system, so AJTCL minimizes the amount of this code that is required to exist on a given device. It does this by constraining the basic environment to a minimal C-only run-time, and by borrowing other devices to perform the router role for it. In contrast to AJSCL, AJTCL, as shown below, does away with much of the overhead present in the AJSL system.



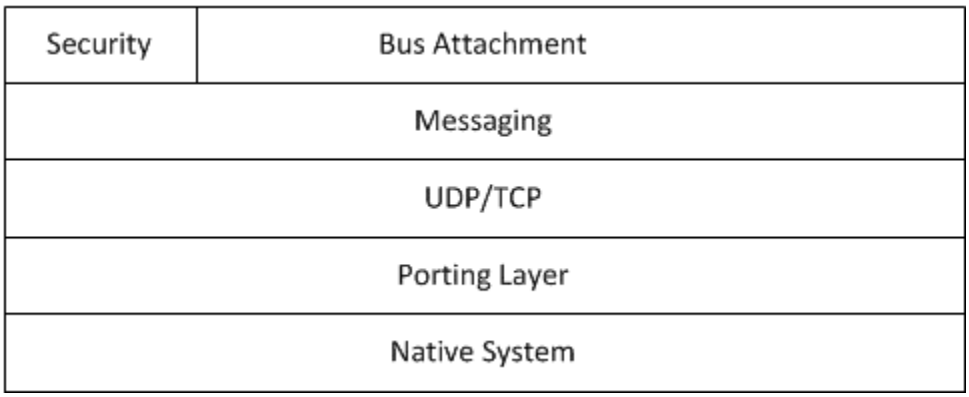


Figure: AJTCL layering

AJTCL exposes only the minimum required API to the bus attachment and exposes the AllJoyn messaging interface directly instead of providing helper functions.

Instead of providing an abstract transport mechanism, the messaging layer uses User Datagram Protocol (UDP) and TCP directly. There is a very thin porting layer to abstract a few needed native system functions, and the entire package is written in C, with an eye toward minimizing code size. Because of these optimizations, an AJTCL system can run in as little as 25 Kbytes of memory, whereas a bundled router and C++ client or service combination may require ten times that amount, and a Java language version may require as much as 40 times that footprint.

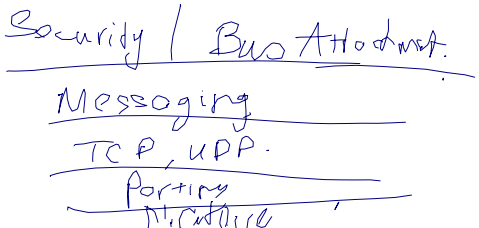
Tying it All Together

In order to make this discussion somewhat more concrete, two example distributed systems are presented here.

- A minimal system in which a single AllJoyn application running on a smartphone talks to a single AJTCL device. This illustrates the trusted router relationship as described above.
- A more complicated system with a router running on a wireless router.

NOTE: Typically, this situation would be a router running OpenWRT that hosts a preinstalled AllJoyn router. This router accepts untrusted connections from Thin Core Libraries that have been onboarded to the Wi-Fi network.

A small number of AJTCL devices connect to the router and act as the sensor nodes for an AllJoyn-based wireless sensor network, and a general purpose computer performs the data fusion function.

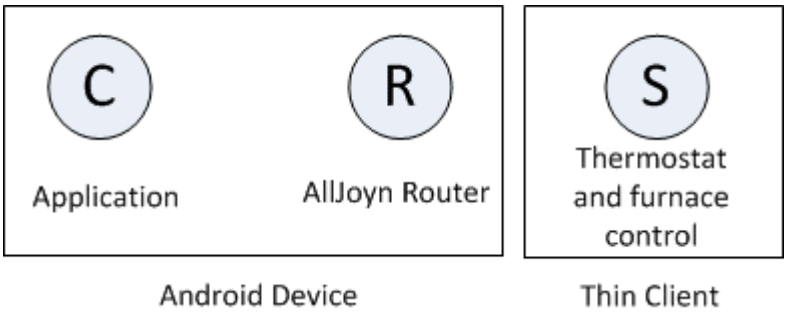


NOTE: In Wireless Sensor Networks, data fusion is a term that refers to a process where some distinguished node collects results from some number of sensor nodes and integrates, or "fuses", its results with those of the other sensor nodes and makes some decision on an action to take as a result of this data.

A minimalist Thin Core Library system

A minimal example of a system using a AJTCL consists of a single host running AJSCL and a Thin Core Library device. AJSCL provides the AllJoyn router which the Thin Core Library will attach to, and also provides a platform for running an application that uses the Thin Core Library. As mentioned above, the Thin Core Library typically acts as a kind of sensor node, and sends data to an application running on the host. The application typically processes the data in some way and issues commands to the sensor to manipulate its environment.

For a plausible but simple system, consider a wall thermostat that controls a furnace, and a control application running on an Android device. The Android device will run AJSCL, and the wall thermostat will run the AJTCL.



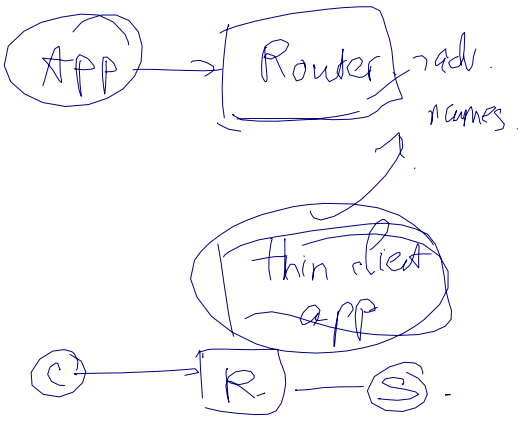


Figure: Minimalist example system

In this example, a requirement is that the wall thermostat only be controllable by a corresponding thermostat controller application in the Android device.

Since a requirement of the example is that the thermostat be controllable only by the Android device, it is probably also a requirement that the thermostat associate itself with only a router associated with the application. This implies that the Android application should be bundled with an AllJoyn router and only this particular combination of bundled router and application should advertise itself as a router for the Thin Core Library to use.

This kind of arrangement leads to a trusted relationship between AJTCL and the router/application pair.

The application then asks its bundled router to quietly advertise a well-known name that is known to AJTCL (for example, com.company.BusNode). The router is then primed to respond to discovery requests for that name in the form of quiet (unicast) responses. When the Thin Core Library comes up, it will perform discovery on the associated network prefix (com.company.BusNode).

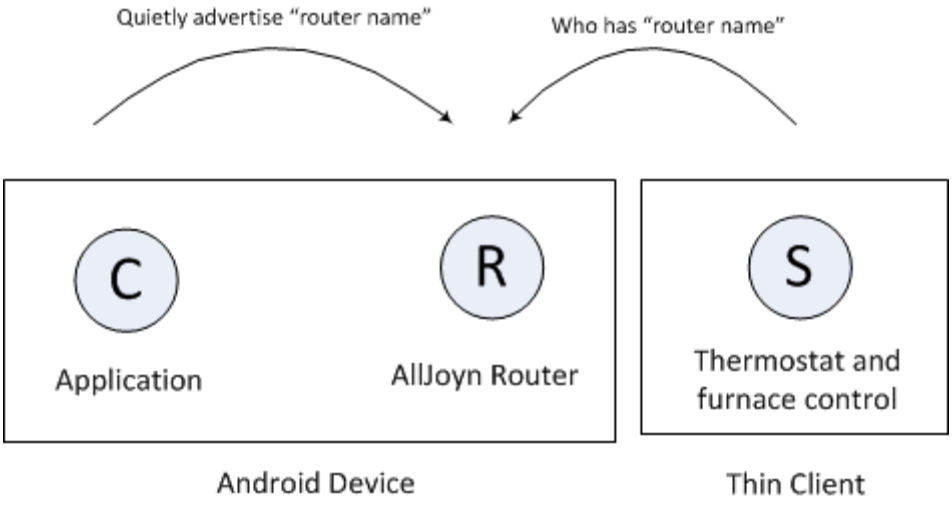


Figure: Thin Core Library router discovery

When the router receives the explicit inquiry about a name it is quietly advertising, it will respond with an indication that the requested name "is at" the particular router. AJTCL will then attempt to connect to the responding router.

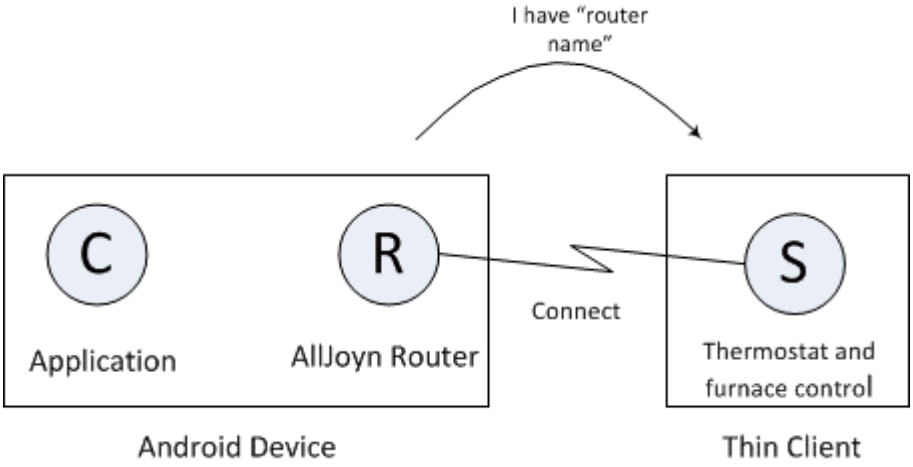


Figure: Thin Core Library connection attempt

At this point, a logical AllJoyn bus has been formed, in which both the application and Thin Core Library service are associated with the bundled router running on the Android device. Representing the system using the bubble diagrams used in **Introduction to the AllJoyn Framework**, the arrangement appears as if the AllJoyn router has a connected service and client.

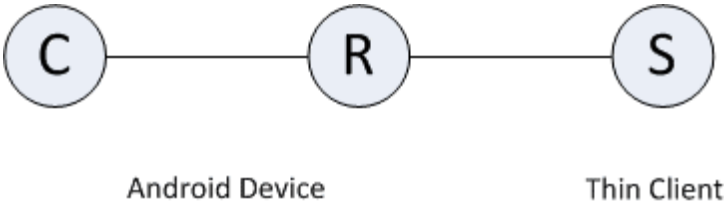


Figure: Thin Core Library system example

At this time, the AJTCL is connected to the router bundled with the application, but neither the application nor the Thin Core Library knows of each other's existence. Typically at this time, AJTCL would request a well-known bus name and instantiate a service in the AllJoyn sense. The Thin Core Library would create a session port and advertise a well-known name as described in **Introduction to the AllJoyn Framework** using the Thin Core Library APIs. This well-known name would typically be different than the well-known name that the bundled router advertises; it corresponds to the client/service relationship between the Thin Core Library and the application, rather than the relationship between the router and the Thin Core Library. The application running on the Android device would then perform service discovery for that name.

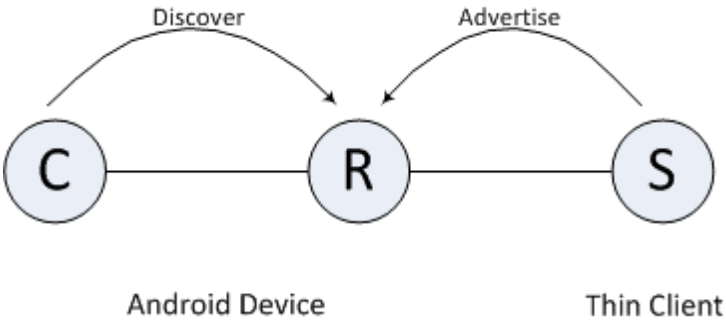


Figure:: Service discovery with the Thin Core Library

When service running on AJTCL is discovered by the client running on the Android device, the client may join the session created by the service.

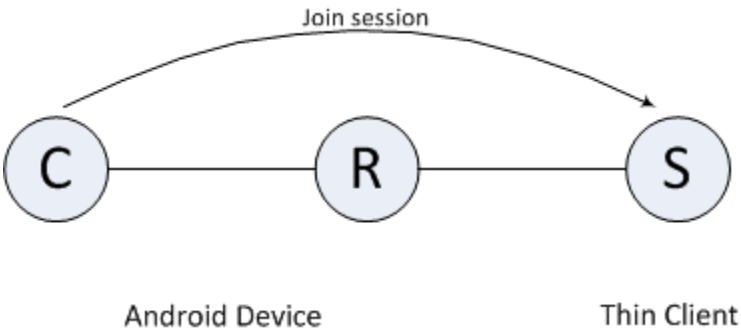


Figure: Android device joins session with service on the Thin Core Library

At this point, the application running on the Android device may access the AJTCL service, as it would any AllJoyn service. It may choose to be notified of signals emitted by the service - in this case, perhaps periodic signals consisting of the current temperature. The application may choose to present a user interface that allows a user to enter a desired temperature and then send that temperature to AJTCL using AllJoyn remote method invocation as described in Introduction to the AllJoyn Framework. Upon receiving a Method Call, the service running in AJTCL could relay the request to the furnace to set the desired temperature.

The API used on the Thin Core Library side is considerably different from that used in AJSCL or a service; however, since the wire protocol is identical in both cases, the flavor of a component on the other side of the connection (AJSCL or AJTCL) is not visible. At this point, AllJoyn is AllJoyn and the bubble diagrams, including AJTCLs, are indistinguishable for all intents and purposes from those bubble diagrams shown in the **Introduction to the AllJoyn Framework**.

A Thin Core Library-based wireless sensor network

This example composes a very basic home management system. The wireless access point is assumed to be an OpenWRT router that hosts a preinstalled AllJoyn router that allows for untrusted Thin Core Library connections. This will allow all AJTCLs participating in the system to connect to the router daemon. Thin Core Library devices in this network could be temperature sensors, motion detectors, light switch actuators, water heater thermostats, furnace or air conditioning system temperature controllers.

As described above, the data fusion function for the example network is performed by an application running on a general purpose computer system with an integrated display. It is not required that there be a dedicated general-purpose computer in the network - data

fusion can be accomplished in a distributed fashion; however, having this component present in the network allows us to illustrate how AJSCL and Thin Core Library devices can interoperate. The "fuser" display could be mounted on a wall in the home or it could simply be the display of a PC located somewhere in the home. This display can, for example, provide user interface elements corresponding to thermometers and thermostats for individual rooms; or virtual light switches, or motion detectors. The actual data fusion function algorithms would determine when to turn lights, home heat, or air conditioner on or off, or when to turn the water heater temperature up or down in the most efficient way.

The first component considered is the OpenWRT router and is illustrated below.

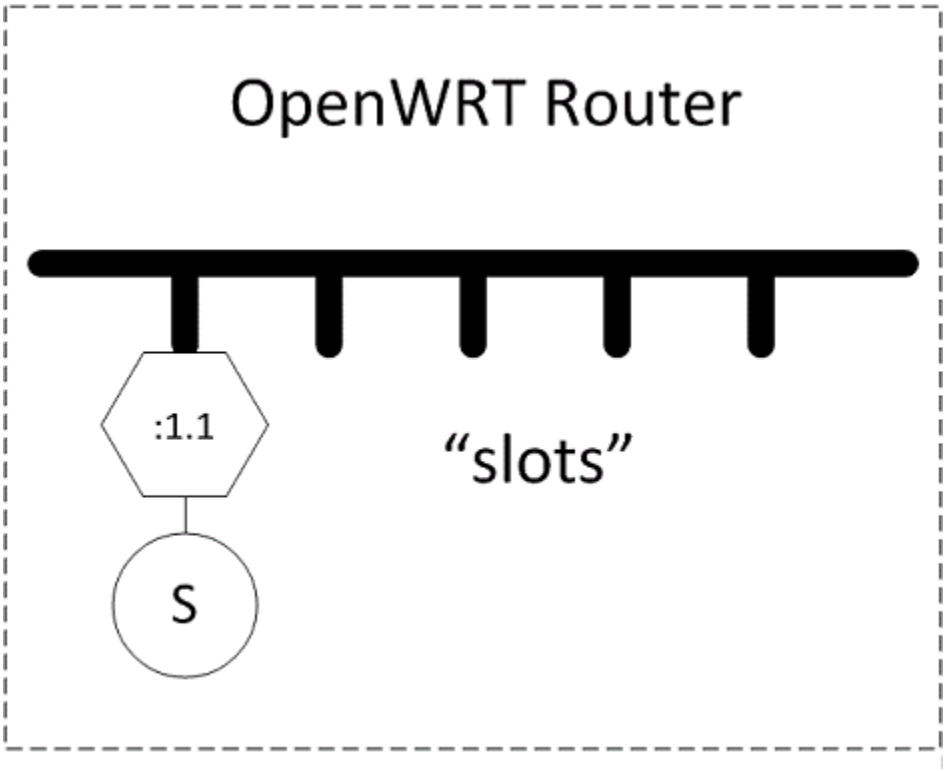


Figure: OpenWRT router hosting a standalone AllJoyn router daemon

The router hosts a standalone AllJoyn router daemon, and is illustrated as the bold horizontal line that represents a segment of an AllJoyn distributed software bus.

There may be an AllJoyn service residing on the router's bus segment that provides a way to configure the router and the preinstalled router using the AllJoyn framework itself. In addition, there are a number of empty slots that represent untrusted connections to

AJTCLs. Since this is a generic AllJoyn router, the corresponding software bus may be extended to other bus segments to form a distributed bus.

As described in the previous section, AJTCL devices will perform discovery to search for a router to which they can connect. Since an untrusted relationship is described here, the AllJoyn router running on the OpenWRT router will be configured to quietly advertise a generic name, perhaps org.alljoyn.BusNode, implicitly indicating that the router is a node on an AllJoyn distributed bus willing to host Thin Libraries.

AJTCLs representing the sensor nodes in the distributed network are brought onto the wireless network through the onboarding process. During this process, they may be assigned so-called friendly names which give them meaning in the context of the home. For example, one light bulb actuator (on-off-dim switch) might be given the name "Kitchen" and another the name "Living Room". The corresponding Thin Core Library nodes begin discovery of their assigned router (perhaps org.alljoyn.BusNode) and will then make connection attempts. Since the slots in the preinstalled router running in the OpenWRT router are presumably untrusted, the Thin Core Library connections are accepted on the network.

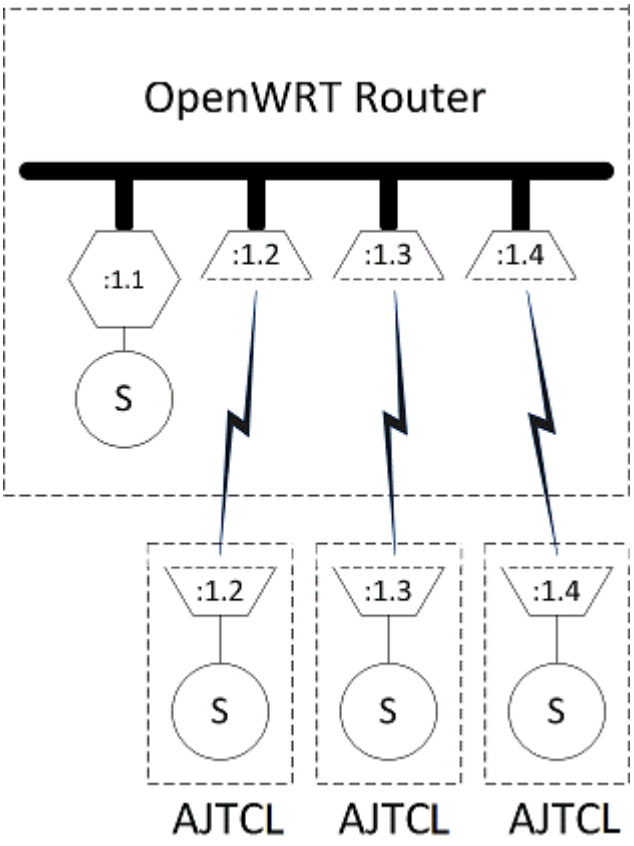


Figure: AJTCL nodes connected to the OpenWRT AllJoyn router

Once the Thin Core Library Apps are connected to the bus segment implemented in the OpenWRT router, they begin to advertise their corresponding services. Presumably, there is also a home control system onboarded to the wireless network provided by the router. This device will be doing service discovery and looking for the service provided by the Thin Core Libraries in the system.

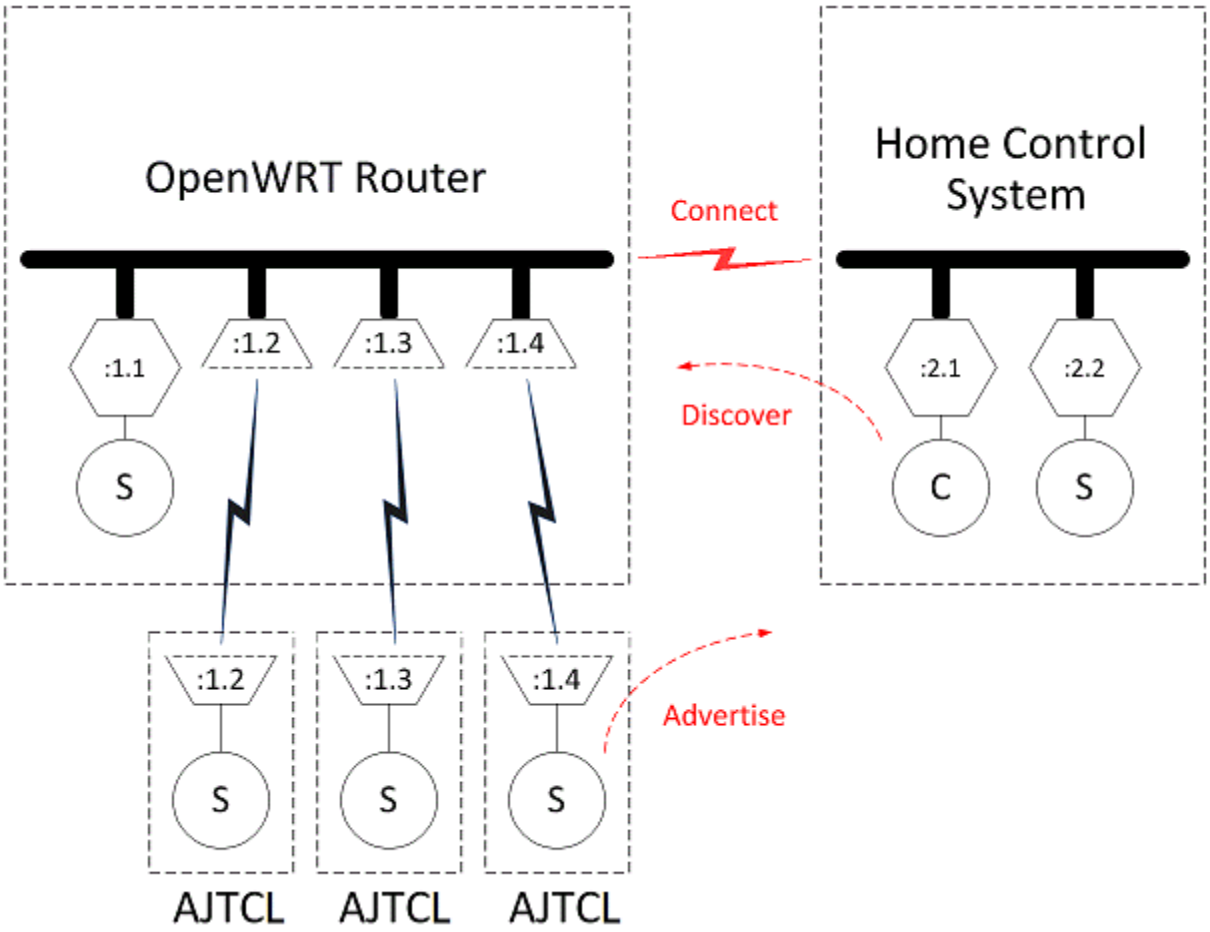


Figure: OpenWRT router, Thin Core Libraries, and home control system

Once the home control system has discovered the service advertisements of one of AJTCLs, it will attempt to join a session with the discovered Thin Core Library as discussed in **Introduction to the AllJoyn Framework**. This will result in the bus segments implemented on the router and the home control system merging into a single virtual distributed bus.

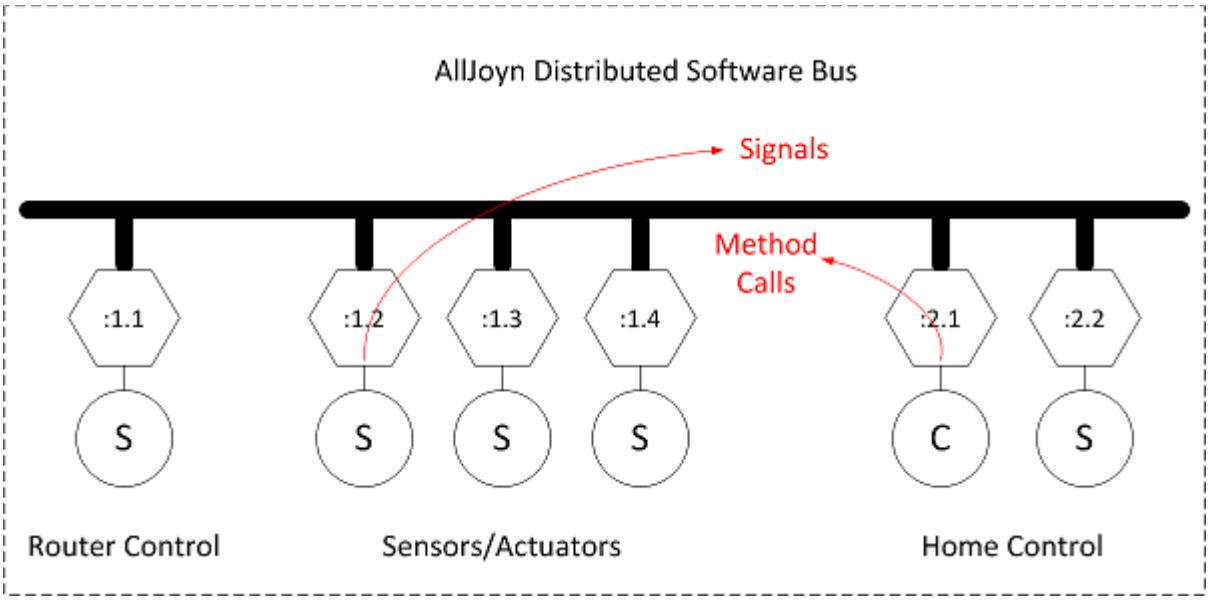


Figure: AllJoyn distributed software bus

When the merged bus is fully formed, the devices attached to the bus behave as generic AllJoyn clients or services. The fact that AllJoyn Thin Core Library sensors and actuators are actually embedded devices connected to an AllJoyn router over TCP is not exposed to other components on the distributed bus. The fact that the home control system is perhaps written in Java and running on a general purpose computer running Android is not exposed to other components on the distributed bus. The clients and services simply make and implement remote method calls and emit and receive signals.

The algorithms running in the data fusion node can now be understood clearly. For example, one important AllJoyn signal sent over the distributed bus might be something corresponding to `CARBON-MONOXIDE-DETECTED`. This signal would be received by the home control system (the data fuser) and it might react by sending a remote method call to one of the actuator nodes telling it to `TURN-FAN_ON`, it might send a remote method call to another actuator node telling it to `SOUND-ALARM`, and it might also send an SMS message to the homeowners letting them know that excess carbon monoxide has been found in the home.

More mundane functions of the home control system might be to make a remote method call to the furnace to reduce the temperature of the home if nobody is present (as reported by motion detectors and a daily schedule). The home control unit may send a message to the water heater telling it to reduce the temperature of the water during the work day or in the middle of the night, but may make a method call to turn the water temperature up in

the middle of the night so that the dishwasher can be run at a time corresponding to the least expensive cost of electricity.

All of the signals that the home control system reacts to and the method calls made are completely independent of the type and location of the source and sink devices.

Summary

AllJoyn is a comprehensive system designed to provide a framework for deploying distributed applications on heterogeneous systems. The AJTCL enables embedded devices to participate in an AllJoyn distributed software bus and present themselves to the rest of the system in such a way as to abstract out the details that usually plague developers in such heterogeneous systems. This approach lets application developers focus on the content of their applications without requiring a large amount of low-level embedded system or networking experience.

The AllJoyn system is designed to work together as a whole and does not suffer from inherent impedance mismatches that might be seen in ad-hoc systems built from various pieces. We believe that the AllJoyn system can make development and deployment of distributed applications that include embedded system components significantly simpler than those developed on other platforms.

Learn More

To learn more about how to integrate the AllJoyn framework in your development efforts, access the documentation and downloads available on the **AllSeen Alliance web site**.

- Introductory guides - Describe AllJoyn technologies and concepts.
- Development guides - Provide guidelines to setting up the build environment and provide solutions to specific programming problems, including code snippets and explanations.
- API references - Provide details for working with the AllJoyn source code and writing applications in each supported programming language.
- Downloads - Software development kits (SDK) provide resources to help users build, modify, test, and execute specific tasks.

Good to know

BASE SERVICES

Base Services are common services used by many devices, providing a set of interfaces for different devices to interact and interoperate with one another. Below are the currently supported base services. This list will continue to expand as more contributions are made to the AllSeen Alliance project.

- **Onboarding**. Provide a consistent way to bring a new device onto the Wi-Fi network.
- **Configuration**. Allows one to configure certain attributes of an application/device, such as its friendly name.
- **Notification**. Allows text-based notifications to be sent and received by devices on the AllJoyn® network. Also supports audio and images via URLs.
- **Control Panel**. Allows devices to advertise a virtual control panel to be controlled remotely.
- **Audio Streaming**. Allows for synchronized audio playback on one or many Sinks.

EVENTS AND ACTIONS API GUIDE

Overview

The AllJoyn® Events and Actions feature enables the ability for the discovery of signals and methods that can be understood by an end consumer in a user interface. An Event is made up of attaching a human readable description to a Signal. An Action is made up of attaching a human readable description to a Method.

This feature/function allows a UI-based application to discover and list Events and Actions in a user interface, and dynamically create a handler for the signal (event) to invoke the method (action) on the respective devices. The information found can also be used to set a rule in an application with the ability to listen for events and perform actions, if an application supports this ability. The rule application provides the best user experience when it is always running and connected to the user's home network.

Architecture

The following figure illustrates one possible basic Action/Event environment that can be built with the Actions and Events feature.

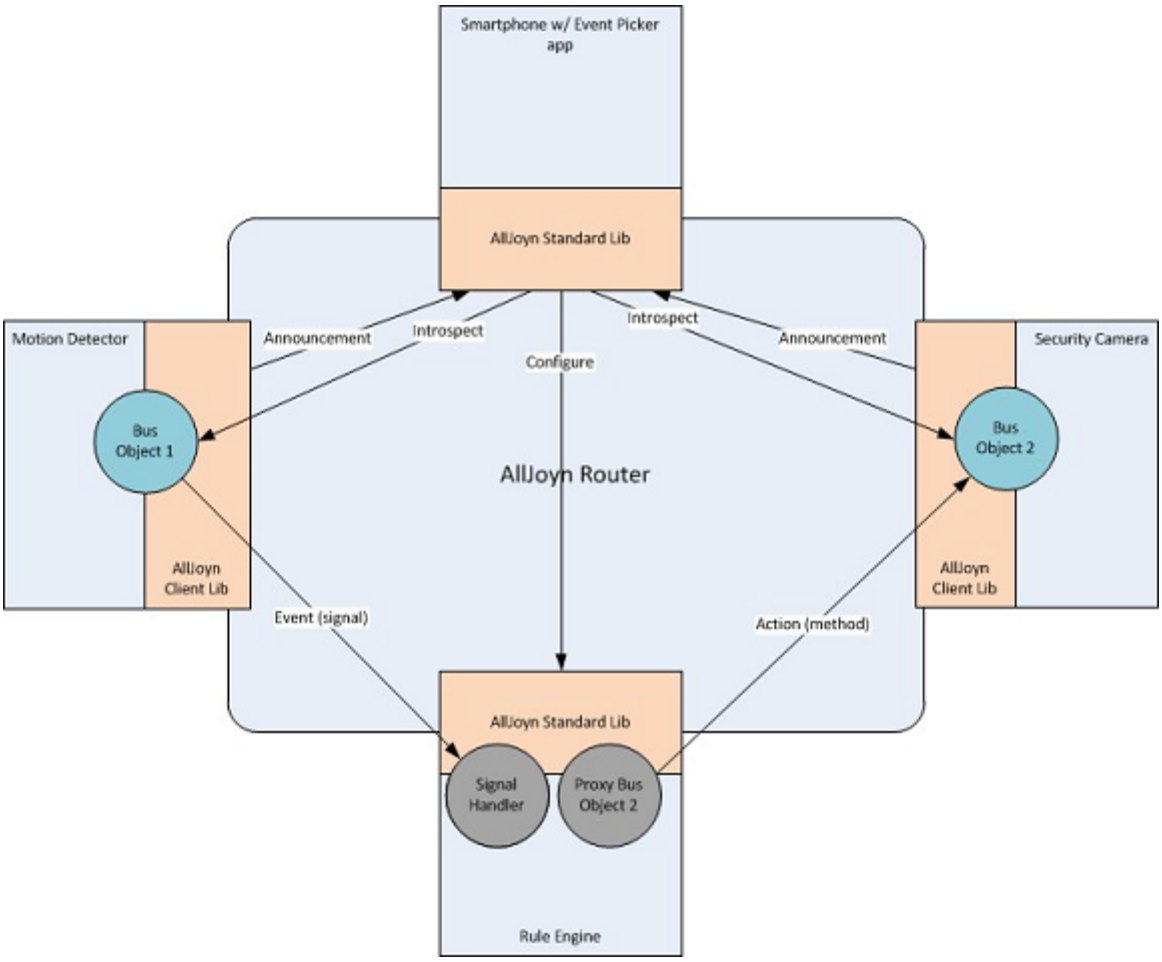


Figure: Events and Actions feature sample architecture

Events and Actions sequence flow

The following steps detail the end-to-end flow involving the Events framework, Actions framework, and Event Picker application. The flow assumes the following:

- Devices that support the Events interface have been onboarded onto the AllJoyn framework and are therefore discoverable on the associated Wi-Fi network.
- Devices that support the Actions interface have been onboarded onto the AllJoyn framework and are therefore discoverable on the associated Wi-Fi network.
- Human readable descriptions have been created as part of the software on the devices to be returned in the introspection XML. These can be created by the OEM, standardized descriptions provided by the AllSeen Alliance can be used.

- A device with the Event Picker application installed discovers other devices on the network that support the Events interface. The device's UI presents a list of those devices such as TV, washer, and thermostat.
- The Event Picker application introspects the Events interface on the devices listed in the UI, and presents a list of human readable descriptions in the UI. Example descriptions include TV is on, Washer cycle complete, and Thermostat cooling.
- The device with the Event Picker application installed discovers devices on the network that support the Action interface. The device's UI presents a list of those devices such as Kitchen Lights.
- The Event Picker application introspects the Action methods on the devices listed in the UI, and presents a list of human readable descriptions in the UI. The events (TV is on, Washer cycle complete, Thermostat cooling) are paired with a corresponding action that can be completed for that device. Examples include Turn light on, Turn light color to blue, Blink light three times.
- The user can select an event and select an action to map the two. This pairs the event and the action, creating a rule the user wants to have occur.

AllJoyn Interface Overview

This section walks through an AllJoyn interface and briefly explains the components. This is not intended to be a replacement for understanding how to build an interface and design one, but just to set the stage on how Events and Actions fit into the core AllJoyn software.

Interface structure

AllJoyn interfaces are defined as XML and contain specific tags that represent the definition expressed. Every interface has a <node> tag that indicates the object path in which the defined interfaces exist. Inside a <node> tag contains the list of interfaces defined in <interface name="SOME_NAME"> tags. These interfaces define the specific functionality that are exposed.

<node> tags can be nested; when no name is supplied, the default is "/".

Methods, Signals, and Properties

Methods, Signals, and Properties make up the definition of the interface for its functionality.

- A method is defined by a tag, Signals and Properties .
- The <method> and <signal> tags allow for <arg> tags to be contained that define the arguments.
- A property does not accept arguments as it is only a single type.

Example interface XML

```
<node>
  <node name="child"/>

  <interface name="com.example.LightBulb">

      <method name="ToggleSwitch">
         <arg name="brightess" type="i" direction="in"/>
      </method>

      <signal name="LightOn" sessionless="true">
      </signal>
      <signal name="LightOff" sessionless="true">
      </signal>

      <property name="LightState" type="y" access="read"/>

   </interface>
</node>
```

org.allseen.Introspectable Interface

To allow the ability to maintain backwards compatibility with DBus, a new Introspection interface was created to allow for the return of descriptions as well as understanding if a signal is sessionless or requires joining a session in order to receive the signal.

Interface org.allseen.Introspectable

The org.allseen.Introspectable interface provides access to the introspection XML containing the description attributes. The following subsection detail the interface's methods and properties.

```
<interface name="org.allseen.Introspectable">
   <method name="GetDescriptionLanguages">
      <arg name="languageTags" type="as" direction="out"/>
   </method>

   <method name="IntrospectWithDescription">
      <arg name="languageTag" type="s" direction="in"/>
      <arg name="data" type="s" direction="out"/>
   </method>
</interface>
```

GetDescriptionLanguages

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 returns ["en", "fr", "cn"].

NOTE: The language tags must comply with IETF language tag standards.

IntrospectWithDescription

This method returns the introspection XML with descriptions in the specified language (exact match only - no best match).

If an element, such as a method, does not have a description in that language, then a language tag determined by the implementer of the introspected device will be used. This provides the following benefits:

- It allows the ability to provide a "default" language so that devices/applications that support Events and Actions will always be visible to a consumer.
- A device manufacturer can provide a closest language algorithm to adjust the default to a language that is close to the requested one.

For example, if a device supports English ("en") and Spanish ("es"), and the requested language is Portuguese ("pt"), it is better to return Spanish as it is Latin-based and may share some common words.

NOTE: The AllJoyn library does not provide IETF language tag matching logic.

Sample introspection XML

The following is a sample of the XML that is returned by IntrospectWithDescription; it adds the description to the LightOn signal which creates a LightOn Event.

```
<node>
   <description>Your lightbulb</description>

   <node name="child">
      <description>Some helpful description</description>
   </node>

   <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
illuminate</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>
```

Setting Descriptions (Standard Core Library)

This section provides usage instructions on the APIs that have been added to the AllJoyn Standard Core Library. These APIs provide the ability to add the introspection descriptions to AllJoyn BusObjects and interfaces. The APIs provide mechanisms for providing these descriptions in multiple languages.

API concepts

As mentioned above, human readable descriptions document BusObjects and interfaces hosted by an AllJoyn service framework. Each description can be offered in multiple languages denoted with an IETF language tag, for example, "en" for English.

Set descriptions for a single language

To provide your descriptions in a single language, use the APIs on the BusObject and interfaces to set the description and its associated language tag.

In all cases where only a single language is set, this string will be returned to an application that asks for descriptions.

Set descriptions for multiple language tags

To provide descriptions in multiple languages provide one or more Translator implementations. The job of the Translator is to translate the descriptions set on the BusObjects and interfaces. Translators may be specific on a BusObject or interface, or global in the BusAttachment.

When generating the introspection XML for each BusObject or interface, the description text is chosen using the following logic:

- The BusObject or interface's own Translator overrides the BusAttachment's Translator.
- A Translator is given priority on providing the translation. For instance, if the BusObject's description is "Turn on the lights" in "en", even if "en" is being requested, the XML generator queries the Translator with "Please translate 'Turn on the lights' from 'en' to 'en' ". IntrospectWithDescription should always return a description in some default language if an unrecognized language is requested. One method of doing this is for the Translator to return NULL, in which case the BusObject or InterfaceDescription's descriptions will be

used as the default, Alternately, the Translator may return a default language translation. More complex algorithms can provide a similar language match so that the end user may have an understanding of the description.

Putting all description texts in a Translator

It is possible to concentrate all descriptions in the Translator implementation by setting the language to the empty string, "", which has a special meaning. It means that the description in the BusObject or InterfaceDescription is not actually a description but rather a "tag" or lookup key that is meant to be passed to a Translator.

In this scenario, it is crucial that the Translator return a default language description even when the language requested is not matched. If the Translator returns NULL, no description will be returned.

NOTE: In this scenario, the description is a lookup tag that is never returned in the introspection XML.

Linux

Setting descriptions in a single language

The easiest way to support descriptions in a single language is to set the description texts on the BusObject and InterfaceDescription objects. Even though the description is only in a single language, the AllJoyn framework still needs to know which language is specified, and the language must be specified in the APIs.

BusObject and InterfaceDescription support setting a single description and its language with the following API methods.

```
void BusObject::SetDescription(const char* language, const char* text)
void InterfaceDescription::SetDescriptionLanguage(const char* language)
void InterfaceDescription::SetDescription(const char* description)

QStatus InterfaceDescription::SetMemberDescription(const char* member,
   const char* description, bool isSessionlessSignal)
QStatus InterfaceDescription::SetArgDescription(const char* member,
   const char* arg, const char* description)
QStatus InterfaceDescription::SetPropertyDescription(const char* name,
   const char* description)
```

Providing translations of descriptions

Defining a Translator

One way to support descriptions in multiple languages is to start with a single description on the BusObject and InterfaceDescription objects and add a Translator. A Translator is an object that implements the ajn::Translator abstract base class.

A Translator class provides the list of languages it can translate to by implementing the following methods which provide the AllJoyn framework with the ability to iterate over the Translator's target languages:

```
size_t NumTargetLanguages()
void GetTargetLanguage(size_t index, qcc::String& ret)
```

In most cases, a Translator provides translations at runtime using the following:

```
const char* Translate(const char* sourceLanguage, const char* targetLanguage,
   const char* source)
```

This function should provide a translation of the "source" text from "sourceLanguage" to "targetLanguage". The string returned must be statically allocated, that is, its memory must not be freed at any point.

Translators returning dynamically allocated strings

In special cases where your application needs to return a string that must be freed at some point after it is used, use this version of the Translate method instead:

```
const char* Translate(const char* sourceLanguage,
   const char* targetLanguage, const char* source, qcc::String& buffer)
```

The "buffer" parameter of this function should hold the memory returned by the function. The following is an example of how to implement this taken from AllJoyn's own code:

```
const char* JTranslator::Translate(const char* sourceLanguage,
      const char* targetLanguage,
      const char* source, qcc::String& buffer)
{
   QCC_DbgPrintf(("JTranslator::Translate()"));
```

```
    /* ... */

    const char* chars = env->GetStringUTFChars(jres, NULL);
    buffer.assign(chars);
    env->ReleaseStringUTFChars(jres, chars);


    return buffer.c_str();
}
```

Setting the Translator

Translators are set on the BusAttachment, BusObject, or InterfaceDescription using the following functions:

```
void BusAttachment::SetDescriptionTranslator(Translator* translator)
void BusObject::SetDescriptionTranslator(Translator* translator)
void InterfaceDescription::SetDescriptionTranslator(Translator* translator)
```

Java

Setting descriptions

To set a BusObject's description, specify it when you register the object:

```
public Status registerBusObject(BusObject busObj,
String objPath, boolean secure, String languageTag, String description)
```

To set descriptions for an interface, use the following annotations:

Interface	Annotation type	Annotations
org.alljoyn.bus.annotation	BusInterface	`descriptionLanguage`, `description`
	BusMethod	`description`
	BusSignal	`descriptionLanguage`, `description`
	BusProperty	`description`

Setting Translators

As in the Linux binding, Translators may be provided on the BusAttachment, BusObject, or Interface levels. The org.alljoyn.bus.Translator interface is equivalent to the Linux ajn::Translator abstract class.

To specify a Translator on the BusAttachment:

```
BusAttachment.setDescriptionTranslator(Translator translator)
```

To specify a Translator on the BusObject with the call to BusAttachment.registerBusObject:

```
public Status registerBusObject(BusObject busObj, String objPath,
boolean secure, String languageTag, String description,
Translator dt)
```

To specify a Translator for an Interface, specify its class name in the BusInterface descriptionTranslator annotation. The AllJoyn framework instantiates a single instance of the Translator class no matter how many interfaces you specify it on.

Objective-C

Setting descriptions

To set descriptions on a BusObject, use AJNBusObject's method:

```
- (void)setDescription:(NSString*)description inLanguage:(NSString*)language
```

To set descriptions on an Interface, use AJNInterfaceDescription's methods:

```
- (void)setDescriptionLanguage:(NSString *)language;

- (void)setDescription:(NSString *)description;
- (QStatus)setMemberDescription:(NSString *)description
forMemberWithName:(NSString*)member sessionlessSignal:(BOOL)sessionless;

- (QStatus)setPropertyDescription:(NSString *)description
forPropertyWithName:(NSString *)propName;
```

```
- (QStatus)setArgDescription:(NSString *)description
forArgument:(NSString *)argName ofMember:(NSStr ing *)member;
```

Setting Translators

The AJNTranslator protocol is equivalent to the C++ ajn::Translator abstract base class. To set the Translator, the following method is supported by AJNBusAttachment, AJNBusObject, and AJNInterfaceDescription:

```
- (void)setDescriptionTranslator:(id<AJNTranslator>)translator;
```

Code generator

The Objective-C code generator supports specifying descriptions in the input XML. Since the XML is modeled on the introspection XML, simply specify the descriptions in the same manner they are returned, but with added attributes to define the language of the descriptions. The <description> element under a <node> or <interface> must contain a "language" attribute.

The following is an example of an XML containing descriptions:

```
<xml>
   <node name="org/alljoyn/Bus/sample">
      <description language="en">This is a sample object</description>
      <annotation name="org.alljoyn.lang.objc" value="SampleObject"/>
         <interface name="org.alljoyn.bus.sample">
            <description language="en">This is a sample interface</description>
         <annotation name="org.alljoyn.lang.objc" value="SampleObjectDelegate"/>

         <method name="Concatenate">
            <description>This concatenates strings</description>
            <arg name="str1" type="s" direction="in">
               <annotation name="org.alljoyn.lang.objc" value="concatenateString:"/>
            </arg>
            <arg name="str2" type="s" direction="in">
               <annotation name="org.alljoyn.lang.objc" value="withString:"/>
            </arg>
            <arg name="outStr" type="s" direction="out"/>
         </method>
      </interface>
   </node>
```

```
</xml>
```

Setting Descriptions (Thin Core Library)

This chapter provides usage instructions on the Thin Core Library usage in order to set descriptions on the interfaces so that events and actions can be discovered.

Thin Linux

There are API calls that allow the ability to assign a description to interface introspection elements. The actual entry of the description is table driven, much like the interface creation. One creates a language tag array, a set of strings they wish to expose as descriptions, a translate function and #defines to lookup the correct strings.

Adding descriptions

Descriptions can be added to any attribute found in the introspection XML. The following subsection detail the ability to return descriptions in the Thin Core Library application.

Create arrays for the language tags used

Depending on the device being made and the target market regions, the developer may wish to support multiple languages. In order to provide this support, create a static array that contains an IETF language tag, for example, "en" for English, "es" for Spanish, etc.

```
static const char* const languages[] = { "en", "es" };
```

Set values for encoded descriptions

Due to running on a memory-constrained platform, the design of providing a description uses an encoding schema. This schema is defined as a uint32 broken up into the following values that are compressed into the single uint32 value:

- BusObject base ID
- Interface index
- Member index

- Arg index

  A helper #define named AJ_DESCRIPTION_ID is found in aj_introspect.h to help encode the messages.

  For example, an encoding for the first AJ_Object would be:

```
AJ_DESCRIPTION_ID(SAMPLE_OBJECT_ID, 0, 0, 0)
```

  An encoding for the first interface in the above object would be:

```
AJ_DESCRIPTION_ID(SAMPLE_OBJECT_ID, 1, 0, 0)
```

Implement Translator method

The Translator method is a function pointer matching the signature of:

```
typedef const char* (*AJ_DescriptionLookupFunc)(uint32_t descId,
const char* lang);
```

This method is invoked on every introspection element and gives the app developer the ability to control what string is set on which element. The implementation of the Translator method should always try and return a description if a descId contains descriptions. This allows an application developer to receive and display descriptions when a lang tag of "" (empty string) or a value that is not contained in the language array are asked for.

NOTE: An example implementation of a Translator is contained in the ajtcl/sample/basic/eventaction_service.c.

Set Translator

Translator methods are set per AJ_ObjectList when the AJ_ObjectList is registered using the C API AJ_RegisterObjetListWithDescriptions.

AJ_ObjectList-specific Translator

To set a specific Translator for an AJ_ObjectList, use the following APIs:

```
AJ_RegisterDescriptionLanguages(const char* const* languages);
AJ_Status AJ_RegisterObjectListWithDescriptions(const AJ_Object* objList,
```

```
uint8_t index, AJ_DescriptionLookupFunc descLookup);
```

First, set the language array using the AJ_RegisterDescriptionLanguages.

Next, register with AllJoyn the AJ_Objects contained in a AJ_ObjectList. Use the method AJ_RegisterObjectListWithDescriptions to both register the AJ_Objects and provide the Thin Core Library with the translator method to invoke when an IntrospectionWithDescriptions call is made on this ObjectList.

Modify interface to specify which signals are Sessionless signals

The character ('&') indicates that a signal will be sent as a Sessionless signal. When a signal is flagged as such and an IntrospectWithDescriptions call is made on the application, it appends the attribute of sessionless=true to the signal introspection tag. For example:

```
static const char* const sampleInterface[] = {
   "org.alljoyn.Bus.eventaction.sample",    /* The first entry
      is the interface name. */
   "?dummyMethod foo<i",    /* This is just a dummy entry at index 0
   for illustration purposes. */
   "?joinMethod inStr1<s inStr2<s outStr>s", /* Method at index 1. */
   "!someSignal",
   "!&someSessionlessSignal",
   NULL
};
```

The interface in the previous example is named org.alljoyn.Bus.eventaction.sample and contains the following:

- Methods
  - dummyMethod with an input variable of foo
  - joinMethod that accepts two strings and returns a string.
- Signals
  - someSignal that requires a session to receive the signal
  - someSessionlessSignal that is sessionless as indicated by the '&' after the '!'.

Adding '&' helps applications decide if they can listen in a passive way (sessionless) or if they must be connected in a session to receive the signal.

Running the Samples

This section describes how to run the core samples in order to test the events and actions feature.

Event picker application for Android

An Application used to find devices that support events and actions is provided as an example. This application lists the device name with the set of actions and events that they expose. It also allows the developer the ability to set up a rule or select a remote rule engine sample application.

Building

The Android application requires the use of the Android NDK. This is done due to the AllJoyn Java language binding lacking the ability to dynamically create and interact with AllJoyn interfaces at runtime.

1. Load the sample application (cpp/samples/eventaction/Android) into Eclipse.
2. Add the Android v4 support library:

a. Right-click on the project.

b. Select **Android Tools > Add Support Library...**.

c. Follow the prompts to download the support library.
3. Open a Windows command prompt and navigate to the /cpp/lib folder.
4. Connect an Android device to the computer.
5. Extract the following library files:

- adb pull /system/lib/libcrypto.so
- adb pull /system/lib/libssl.so
6. Navigate to the eventaction sample, /cpp/samples/eventaction/Android.

7. Build MyAllJoynCode.so using the Android NDK:

- ndk-build

8. Go back into Eclipse and run the application on your mobile device.

. Right-click on the project.

a. Select **Refresh**.

b. Right-click on the project again.

9. Select **Run As > Android Application**.

Usage

The Event Action Browser (EAB) uses the AllJoyn About feature to find devices and performs an IntrospectWithDescription request. If any descriptions are found, they display as an Action or Event under the corresponding section.

Long pressing on a found item displays more developer information such as the busName, path, interface, etc.

The EAB further allows for the creation of a rule. Only one Event can be selected and one or more Actions can be selected. Once checked, select **Save Rule**. This now causes the EAB to listen for the Signal and make BusMethod calls on the actions.

The application is purely an example and does not allow the viewing or deletion of a single rule. Pressing **Delete Saved** will clear out the saved instances.

If a remote Sample Rule Application is running, the drop-down at the top of the EAB lists the names of the Rule instances. The save and delete functions the same way, only now another device is going to listen for the Signals and place BusMethod calls.

Standard Core Library

Linux sample service

Descriptions have been added to the sample in alljoyn_core/samples/basic/signal_service.cc.

Compile

The sample compiles together with alljoyn_core.

Run

Run the application at the following location:

```
build/<os>/<cpu>/<debug|release>/dist/cpp/bin/samples/signal_service
```

Java sample service

Sample service and client applications can be found at the following locations:

- samples/java/JavaSDKDoc/JavaSDKDocIntrospectWithDescriptionService/
- samples/java/JavaSDKDoc/JavaSDKDocIntrospectWithDescriptionClient

Compile

The samples compile together with alljoyn_java.

Run

To run the service:

```
java -jar
build/<os>/<cpu>/<debug|release>/dist/java/jar/JavaSDKDocIntrospectWithDescriptionSer
vice.jar
```

To run the client:

```
java -jar
build/<os>/<cpu>/<debug|release>/dist/java/jar/JavaSDKDocIntrospectWithDescriptionCli
ent.jar
```

Thin Core Library

Compile

1. Navigate to the ajtcl folder.

2. Type `scons` to build on the platform.

Run

The eventaction_service application executable is found under ajtcl/samples/basic/. Run this application to exercise the ability to offer descriptions when an IntrospectionWithDescriptions call is made.

```
$ ./samples/basic/eventaction_service
```

Best Practices and Common Issues

What should a Translator method do?

The Translator method should always return a human readable string regardless of the asked for language. This allows for an end user to always discover the Event and Action descriptions to be shown in a UI.

The Translator should first try and match the requested language tag to one that is supported and return the correct string. At a minimum, if the language tag does not match it should return a string in a default language. The default language is up to the creator of the device to determine.

When to use Events and Actions?

The power of the Events and Actions feature is that a small change helps enable an end consumer to connect applications together in new ways. When a device supports certain features and functions that are interesting, they should be considered as being expressed as an Event or Action

For example, a talking teddy bear can have a series of phrases that can be spoken. The bear by itself can say things like "Good morning", etc. By simply creating the ability for GoodMorning to be expressed as an Action with a description of "Say Good Morning", the bear can now be connected to other devices. When a consumer has a smart alarm clock, the clock can be connected to have the bear speak the "Good Morning" greeting to wake them up.

Most devices/applications have a set of defined features. To allow for greater reach and interactions, these features should be evaluated to understand if a simple presentation to an end user creates a larger experience with other devices.

Should I introspect everything?

Since any device can contain events or actions, it should be introspected to check for these. However, it is recommended first to use the GetDescriptionLanguages API call to do the following:

- Check that the "org.allseen.Introspectable" interface is present.
- Ensure that the language tag exists that matches what the consumer has set on a mobile device.

  If the language tag does not match, a pop-up or feedback to the user can ask the user what language they would like descriptions in, else the default language will be returned. If GetDescriptionLanguages returns an error, this means that the device has no events or actions and should not be introspected again.

  When the introspection with descriptions is made, a commercial quality application should store the object path and device information so that only an introspection request on specific objects on devices can be made in the future. This allows for less traffic over the network

Which language tag is used when introspecting?

It is recommended to use the GetDescriptionLanguages API so that the consumer knows in advance what language will show up from the device. Since Events and Actions really revolve around the end consumer, it is important to provide them with information that makes sense. It is best to use the language from the device settings, however, the application may want to present a user with a list of languages to pick from.

As a failsafe, if a device is introspected with an invalid tag, it should return a default description. Per the information in **What should a Translator method do?**, a device or application should always return a description that can be rendered in a UI for a consumer to link events and actions.

Can the Rule engine be used as is?

The sample rule engine is very primitive and sets up a link between devices. In the AllJoyn SDK, the rule engine is currently contained in both an Android application and as a standalone process. In the sample rule engine, the linkage between and event and action (rule) is not long-lasting. When an application/device powers on and connects to the AllJoyn network, it changes the busName. This means that the saved information in the rule is no longer valid. A commercial quality Rule Engine saves off the DeviceId and ApplicationId from the About Announcement, and then updates the saved busName with what was received in the About Announcement. This allows for devices to be powered off and come back on later and have the rules still apply.

The sample rule engine also does not provide the ability to list the saved rules or edit rules. This is something that is out of scope for the sample application.

As a standalone rule engine, the communication to the Android sample is through an AllJoyn interface and is done as a sample, not a final solution.

- **Base Services**
- **Glossary**

ALLJOYN® SYSTEM DESCRIPTION

Release history

Release version	Date	What changed
14.06	9/26/2014	Initial release
14.12	12/17/2014	Updates for new functionality added in 14.12 release: UDP Transport design TCP vs UDP Transport selection logic at the router mDNS-based discovery for the router at the TCL Updates to SLS fetch backoff design to support linear+exponential backoff Router Probing mechanism to detect missing applications Router logic to detect and disconnect slow reader nodes Other updates: Endpoints usage by AllJoyn Transport TCP Transport data plane architecture and state machine AllJoyn Protocol Version mapping for different releases Link timeout mechanism between routers to detect missing routers
15.04	4/29/2015	Updates In Thin Apps section related to: Security and adding description of Router Selection Some general clean up including fixing typos and readability and consistency changes Other changes: General cleanup

Release version	Date	What changed
		Removed references to RSA and PIN authe mechanisms as they are not longer supported

This section describes in detail how AllJoyn works at the system level.

System Overview

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.

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.

The following figure shows an example IoE network with multiple proximal IoE networks interconnected with each other via the Internet.

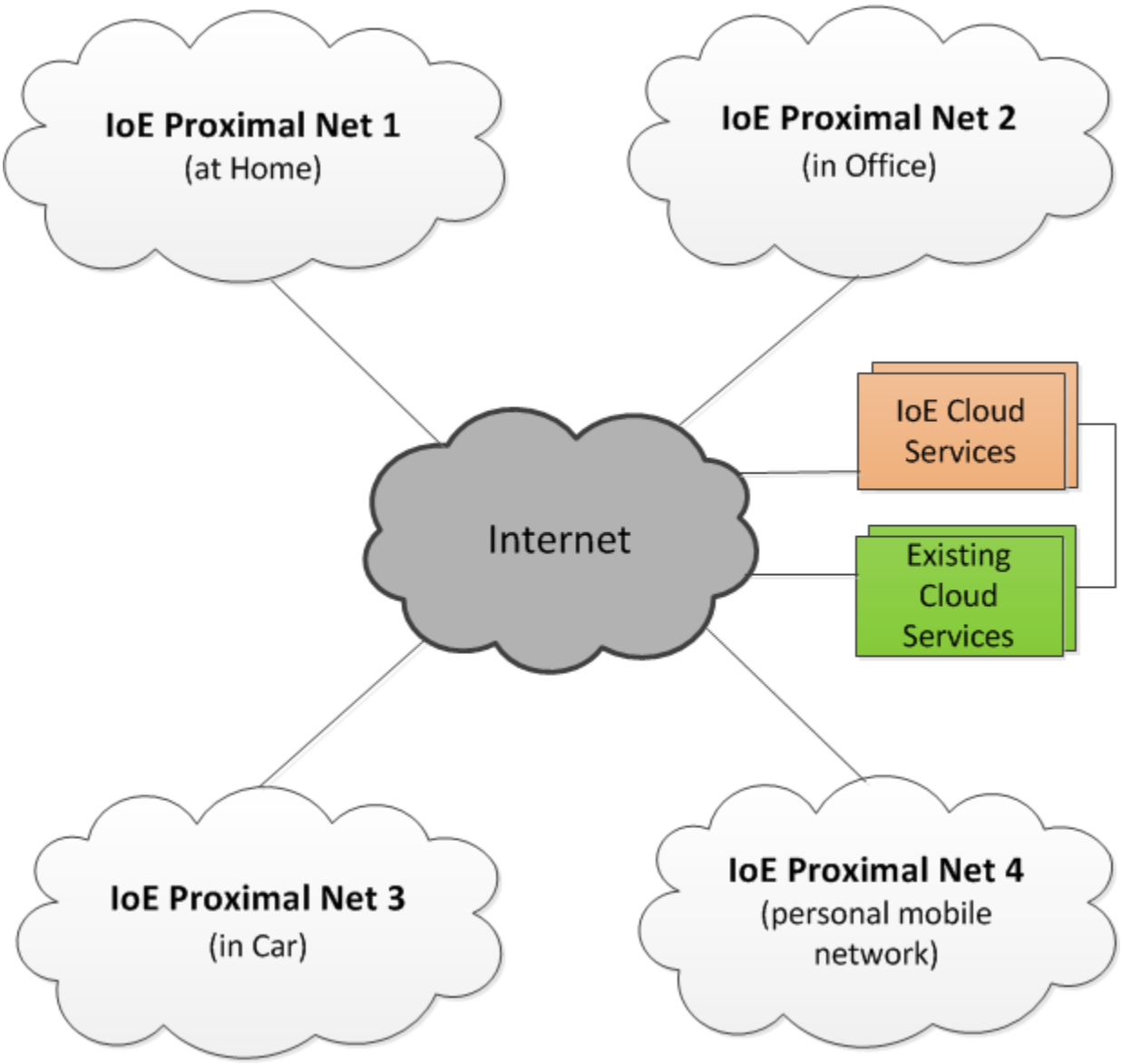


Figure: IoE network example

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.

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.

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.

The following figure shows an AllJoyn network with 4 devices.

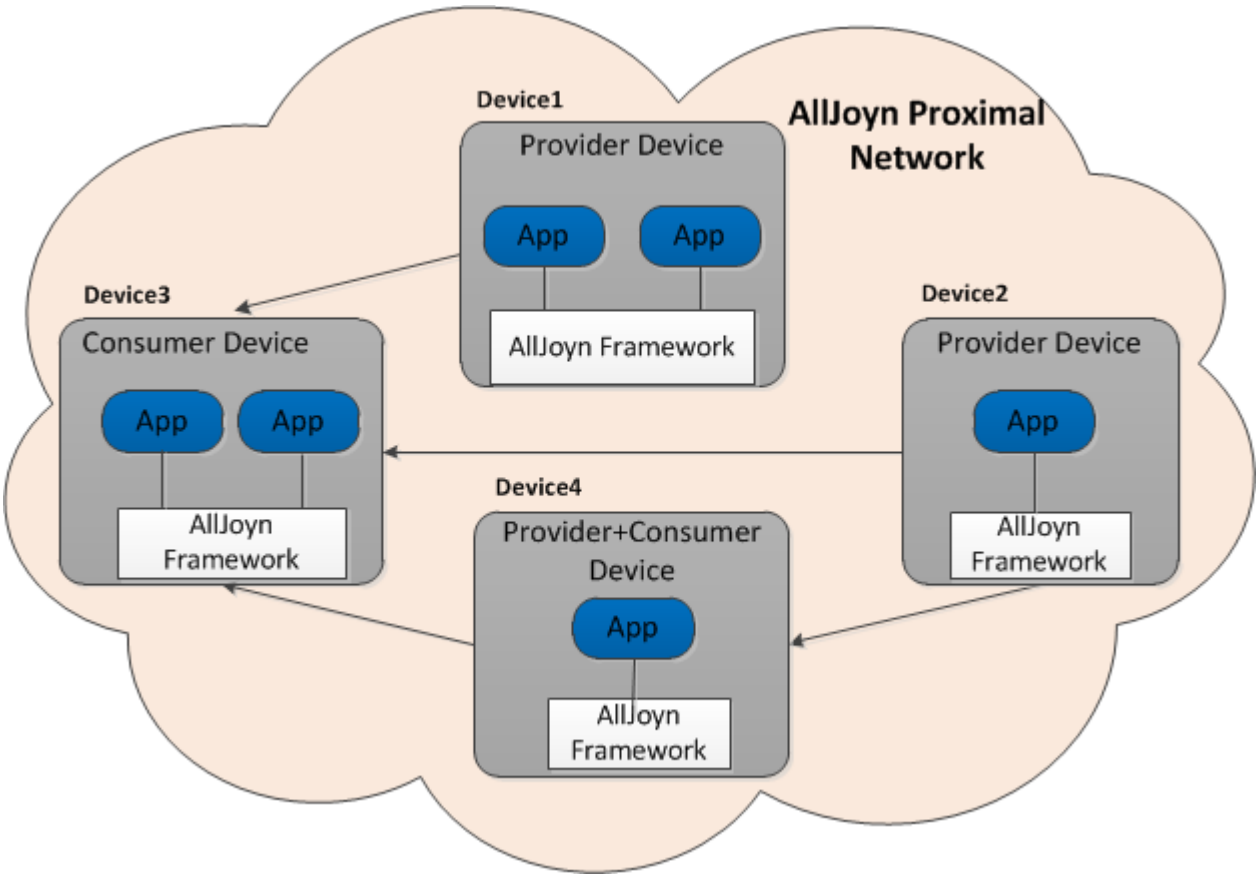


Figure: AllJoyn network

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 that 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.

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 2.3 and later, common versions of Microsoft Windows OS, Apple iOS, Mac 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.

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 **Message format**).
- 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**).

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.

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 below.

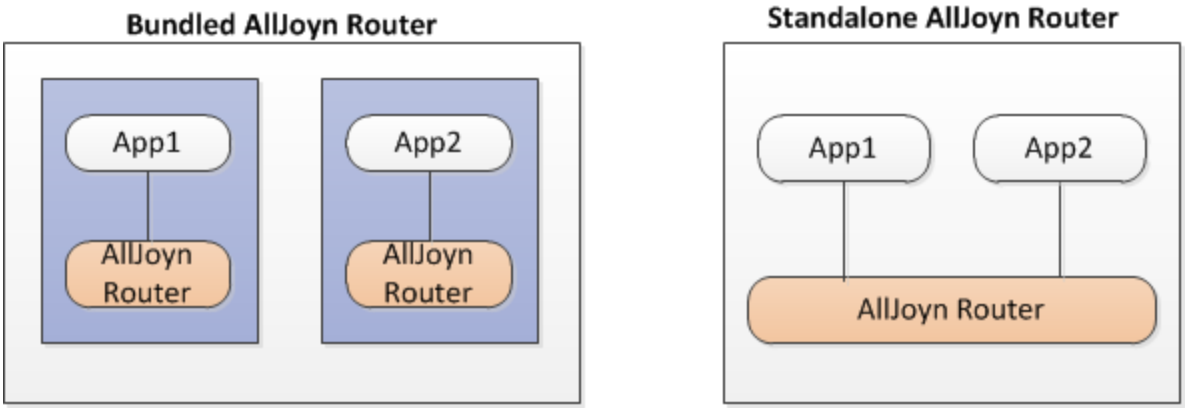


Figure: AllJoyn bundled and standalone router examples

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.

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 below.

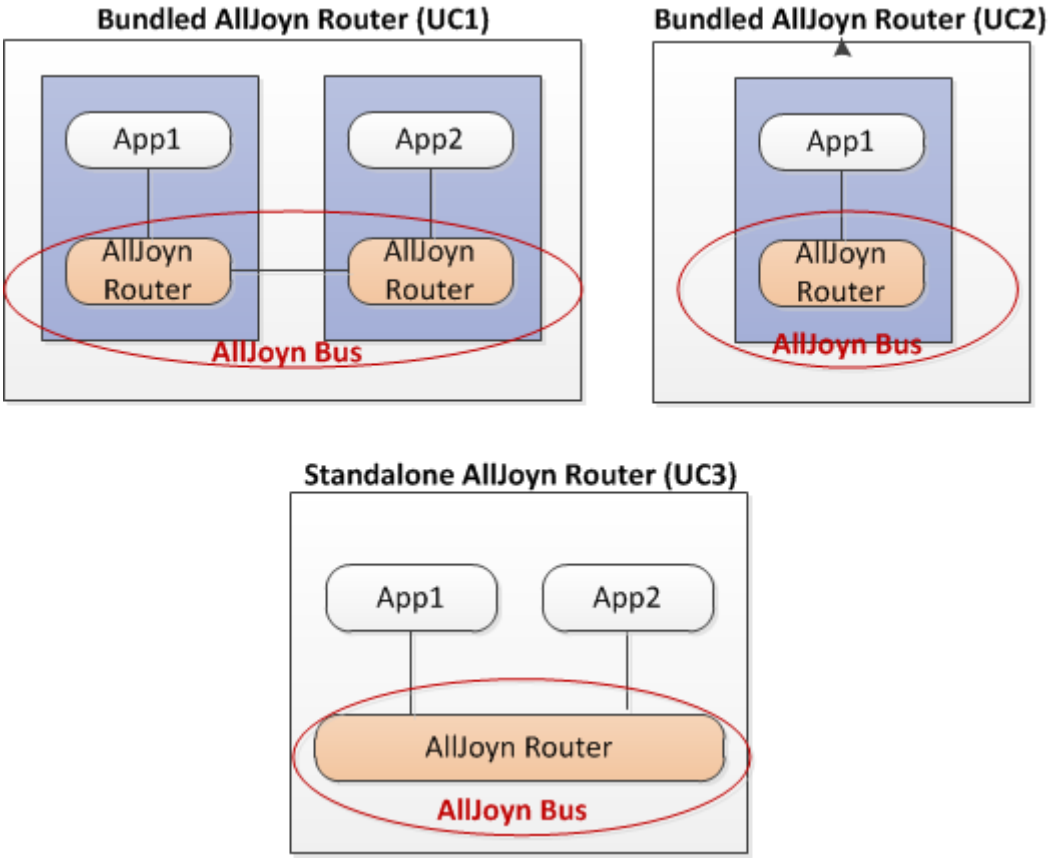


Figure: Logical mapping of AllJoyn router to AllJoyn bus

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.
- Standalone deployment model with one or more apps on the device, shown as UC3.

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

**NOTE:*8 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.

The following figure shows a simplistic view of the local AllJoyn bus on two different devices with multiple applications connecting to the bus.

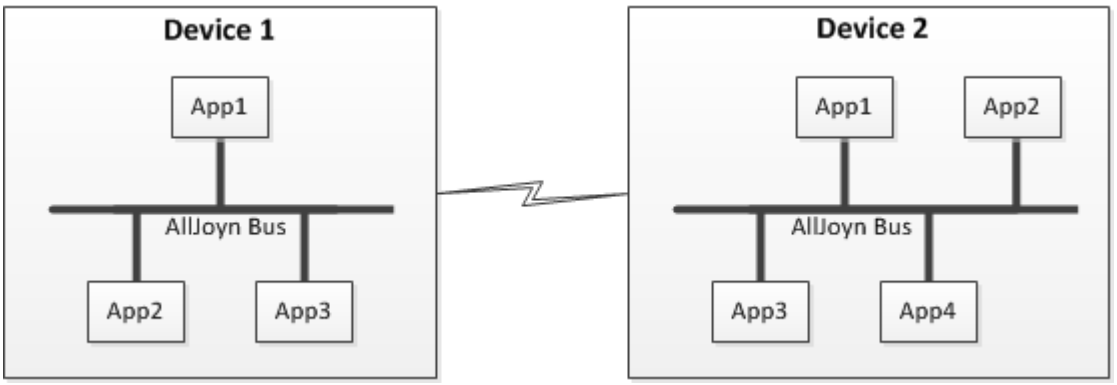


Figure: AllJoyn bus

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.

Multiple instances of AllJoyn buses across multiple devices form a logical distributed AllJoyn software bus as shown below.

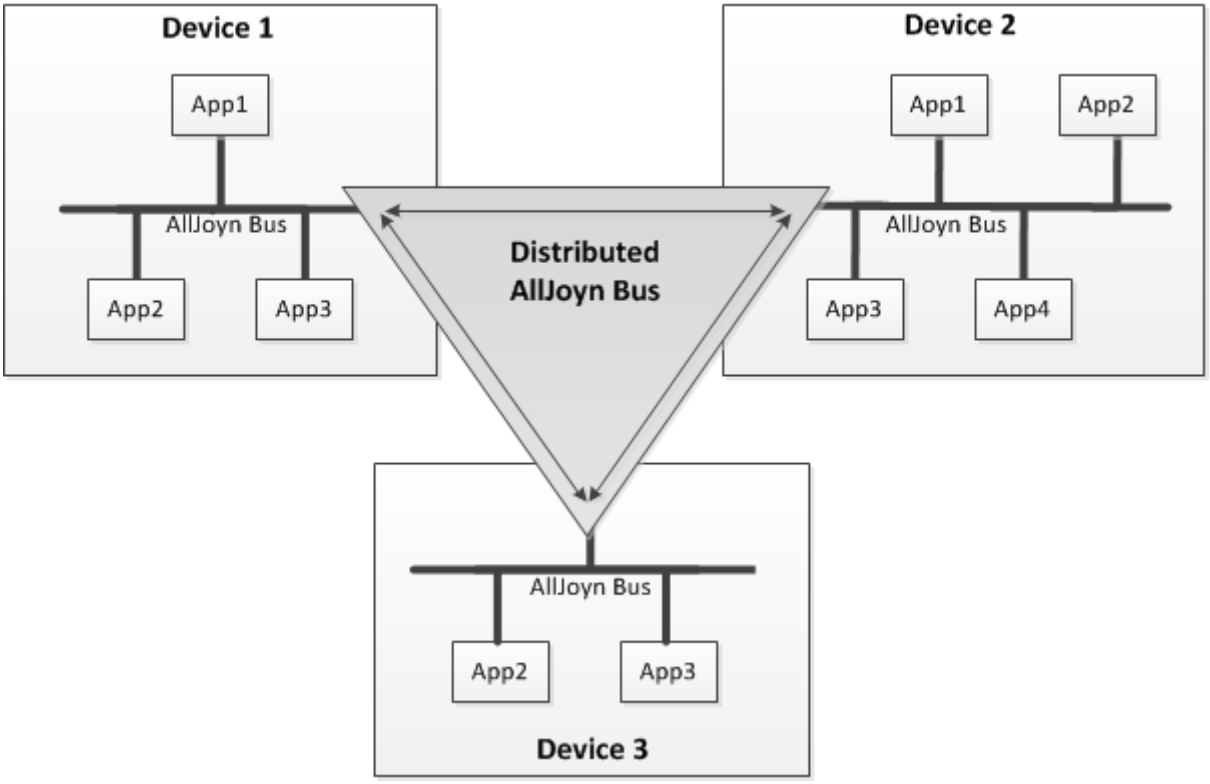


Figure: Distributed AllJoyn bus

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.

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 **AllJoyn interfaces**) 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.

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 ":.1" unique name is always given to the AllJoyn router local endpoint.

The following figure shows the unique name assignment for three connected apps to an AllJoyn bus by a single AllJoyn router with GUID=100.

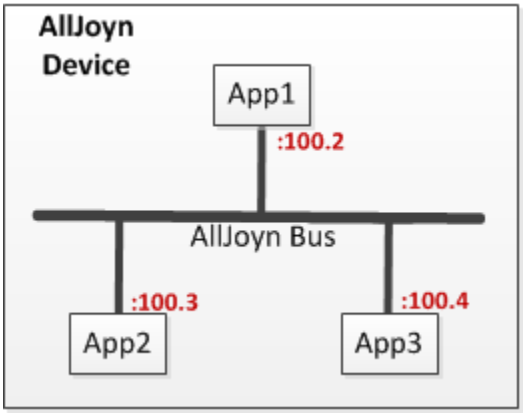


Figure: AllJoyn unique name assignment 1 (multiple apps connected to single AllJoyn router)

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.

The following fiture shows the unique name assignment for AllJoyn apps with multiple instances of an AllJoyn router forming an AllJoyn bus.

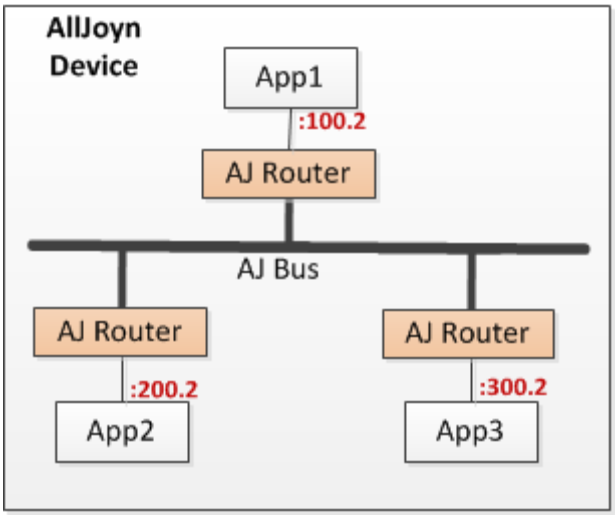


Figure: AllJoyn unique name assignment 2 (each app has instance of AllJoyn router)

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

The following figure shows the unique name assignment for AllJoyn apps on two different devices connected over a distributed AllJoyn bus.

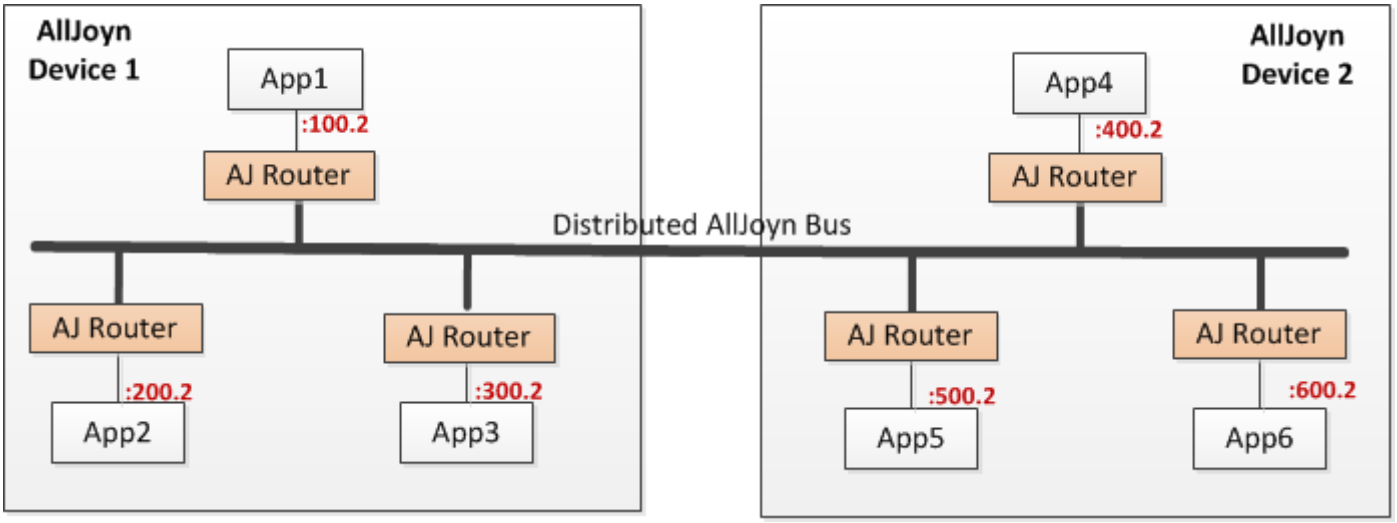


Figure: AllJoyn unique name assignment 3 (AllJoyn apps on two devices connected over distributed AllJoyn bus)

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
```

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.

The following figure shows the distinction between the AllJoyn service object and proxy object.

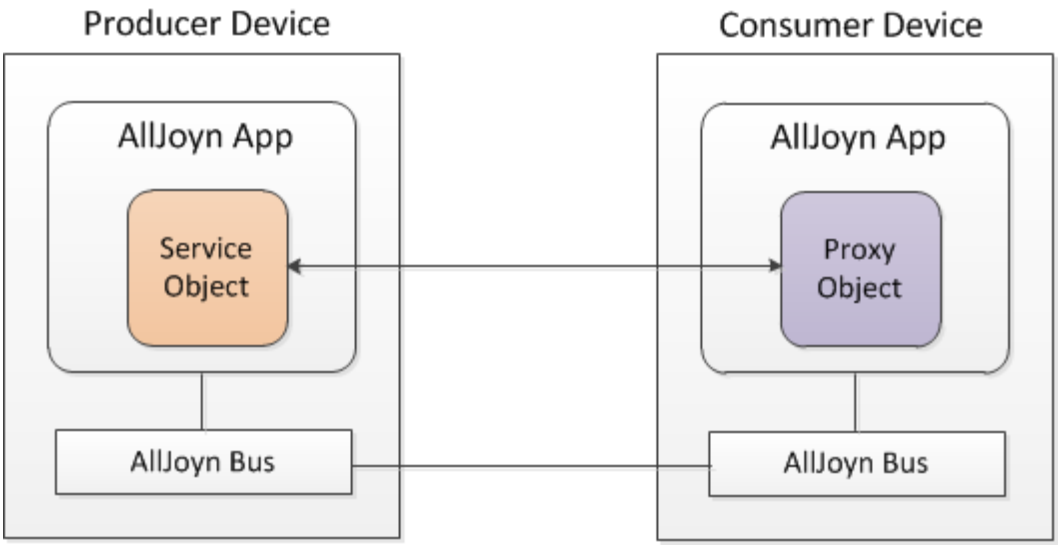


Figure: AllJoyn service object and AllJoyn proxy object

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
```

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 interoperability 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
```

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.

The following figure shows three apps connecting to a given AllJoyn bus via the AllJoyn core Library.

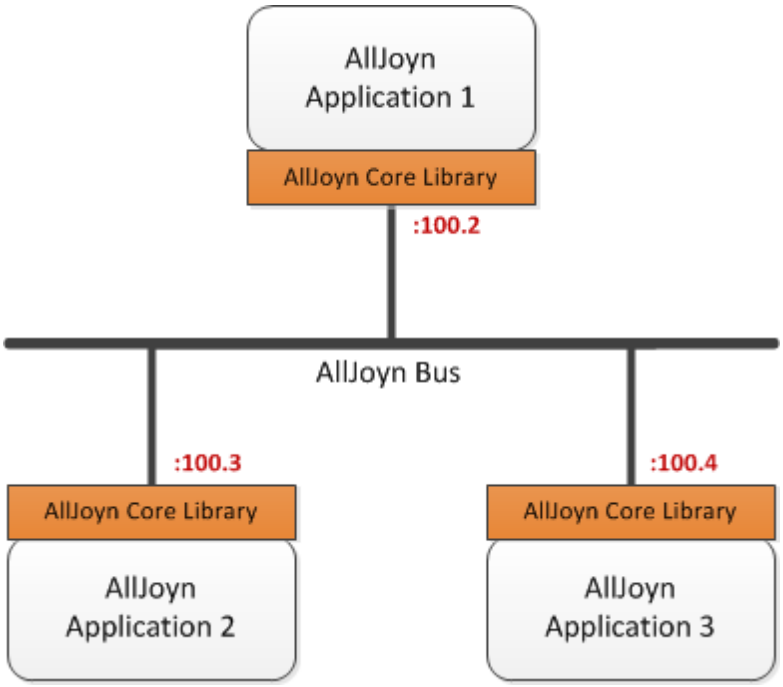


Figure: AllJoyn core library

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 **Thin Apps**.

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. See the [About HLD] for design details on the About feature.

AllJoyn endpoints

AllJoyn applications exchange data in the form of D-Bus formatted messages. These messages specify source and destination as Endpoints. An AllJoyn Endpoint represents one side of an AllJoyn communication link.Endpoints are used to route messages to appropriate destinations.

Both the Core Library and AllJoyn Router maintain endpoints to enable message routing. The Core Library maintains the following endpoints:

- **Local Endpoint**: The local endpoint within the Core Library represents a connection to the attached application.
- **Remote Endpoint**: The remote endpoint within the Core Library represents the connection to the AllJoyn router.This is applicable only for the case when AllJoyn router is not bundled.

An endpoint maintained by the AllJoyn router is uniquely identified by a unique name assigned to it.The AllJoyn router supports the following 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 ":.1"

- **Remote Endpoint**: A remote endpoint identifies the connection between the application and the AllJoyn router. Messages destined to applications get routed to app endpoints.
- **Bus-to-Bus Endpoint**: A Bus-to-Bus (B2B) endpoint is a specialized kind of remote endpoint that 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.

The following figure shows different endpoints in the AllJoyn system.

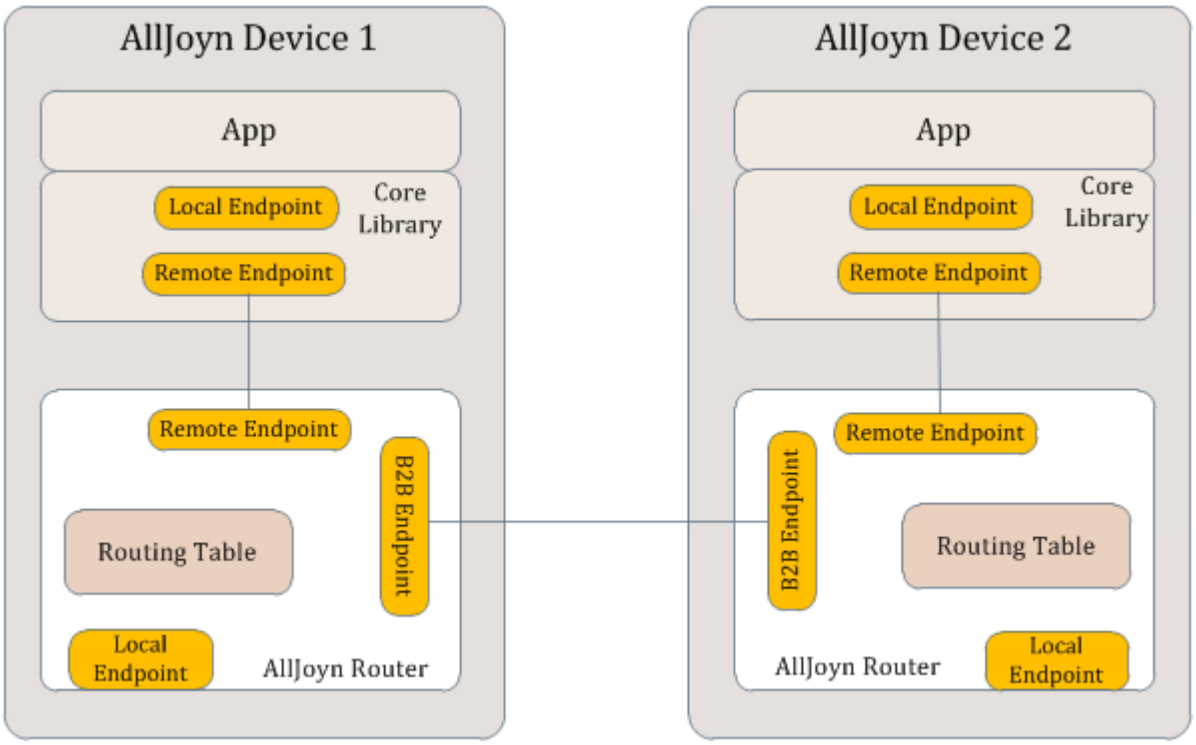


Figure: AllJoyn endpoints

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.

AllJoyn entity relationship

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

The following figure captures the relationship between various high-level AllJoyn entities including device, application, objects, interfaces, and interface members.

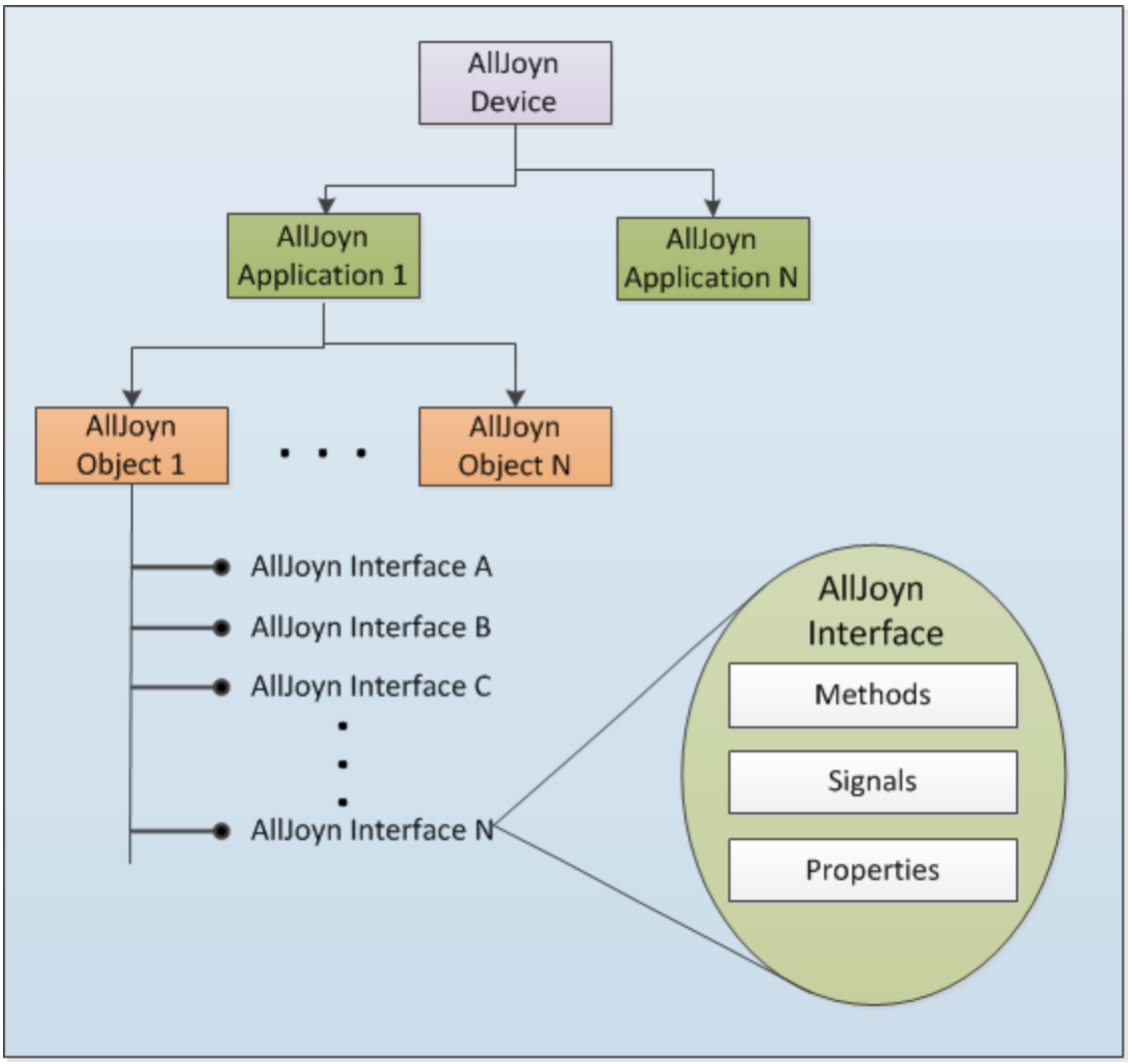


Figure: AllJoyn entity relationship

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.

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 the following figure.

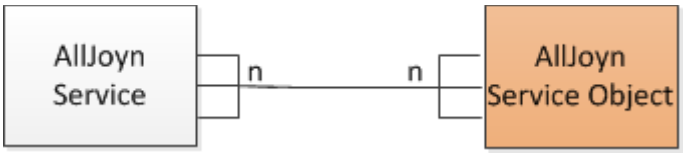


Figure: AllJoyn service and AllJoyn service object relationship

AllJoyn services

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

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.

NOTE: 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.

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.

AllJoyn transport

The AllJoyn Transport is an abstract concept that enables connection setup and message routing across AllJoyn applications via AllJoyn routers. The AllJoyn transport logic in turn supports transmitting messages over multiple underlying physical transports including TCP transport, UDP transport and Local Transport (e.g., UNIX domain sockets).

The AllJoyn transport logic delivers the advertisement and discovery messages based on specified list of transports by the app. Similarly, the AllJoyn transport enables session establishment and message routing over multiple underlying transports based on transport selection made by the application. The set of underlying transports supported by the AllJoyn transport is specified by a TransportMask as captured in **AllJoyn Transport in Networking Model**. If an app does not specify any transport(s), the AllJoyn transport value defaults to TRANSPORT_ANY.
See **AllJoyn Transport** for more information.

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.

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 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.

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 **Sessionless Signal**).

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.

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.

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.

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.

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 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 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.

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.

AllJoyn protocol version

Functionality implemented by the AllJoyn Router is versioned through an AllJoyn Protocol Version (AJPV) field. The following table shows the AJPV for various AllJoyn releases; unless otherwise noted the AJPV for the major release version applies to all the patch release versions as well. The AJPV is exchanged between routers as part of the BusHello messaging during the AllJoyn session establishment and between the leaf and routing node when the leaf node connects to the router. This field is used by the core libraries to identify compatibility with the router, and specifically by thin apps to determine whether or not to connect to a particular router or keep searching for another one. It is also used by the router to determine if functionality is available at the leaf (e.g. self-join, SessionLostWithReason, etc.)

Table: AllJoyn Release to Protocol Version mapping

Release version	AJPV
legacy 03.04.06	9
v14.02	9
v14.06	10
v14.12	11
v15.04	12

ABOUT FEATURE INTERFACE DEFINITIONS

Release History

To access a previous version of this document, click the release version link below.

Release version	Date	What changed
14.02	2/28/2014	About interface version 1 was added.
14.06	6/30/2014	No updates.
14.06 Update 1	9/29/2014	Updated the document title and Overview chapter title (changed from Specification to Definition) Added the release version number to the document title for version tracking. Added a note in the Definition Overview chapter to address the AllSeen Alliance Compliance and Certification program. Added a Mandatory column for method and signal parameters to support the AllSeen Alliance Compliance and Certification program.
14.12	12/17/2014	Changed DeviceName from required to not required Additional clarification specifying the AppId must be 128-bit UUID as specified in RFC 4122 Cleanup to make requirements for methods and signals more clear Icon interface was added. The icon interface has been part of AllJoyn™ and the About Feature since 14.02; however, the interface definition documentation was not added until 14.12.

Definition Overview

The About interface is to be implemented by an application on a target device. This interface allows the app to advertise itself so other apps can discover it. The following figure illustrates the relationship between a client app and a service app.

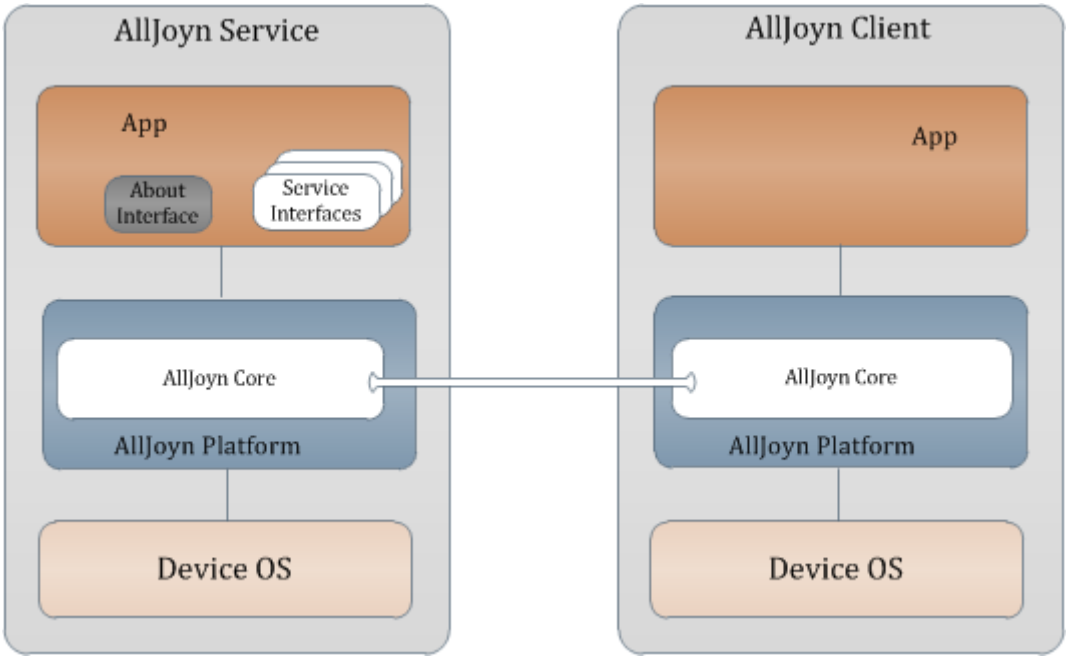


Figure: About feature architecture within the AllJoyn™ framework

NOTE: All methods and signals are considered mandatory to support the AllSeen Alliance Compliance and Certification program.

Discovery

A client can discover the app via an announcement which is a sessionless signal containing the basic app information like app name, device name, manufacturer, and model number. The announcement also contains the list of object paths and service framework interfaces to allow the client to determine whether the app provides functionality of interest.

In addition to the sessionless announcement, the About interface also provides the on-demand method calls to retrieve all the available metadata about the app that are not published in the announcement.

Discovery Call Flows

Typical discovery flow

The following figure illustrates a typical call flow for a client to discover a service app. The client merely relies on the sessionless announcement to decide whether to connect to the service app to use its service framework offering.

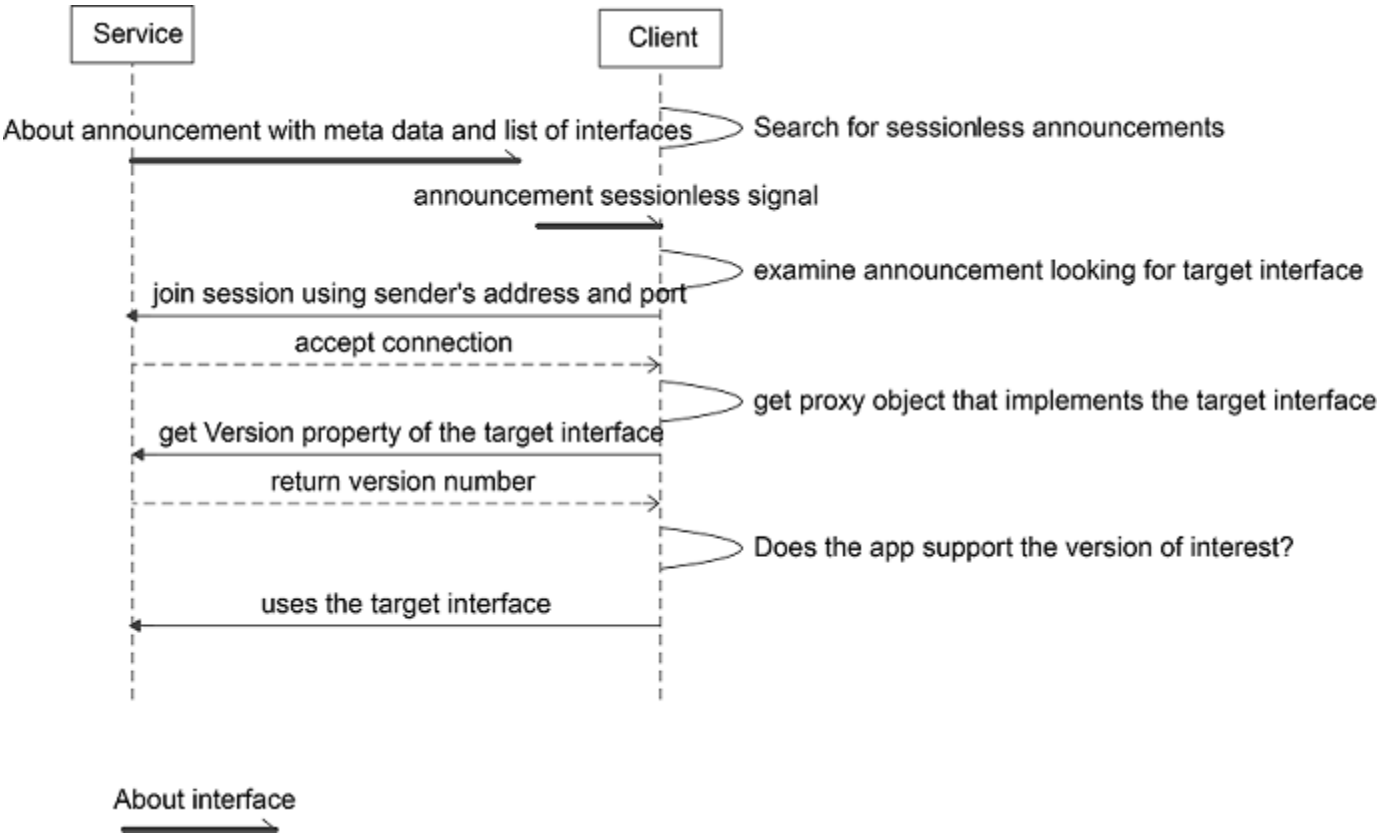


Figure: Typical discovery flow (client discovers a service app)

Nontypical discovery flow

The following figure illustrates a call flow for a client to discover a service app and make a request for more detailed information.

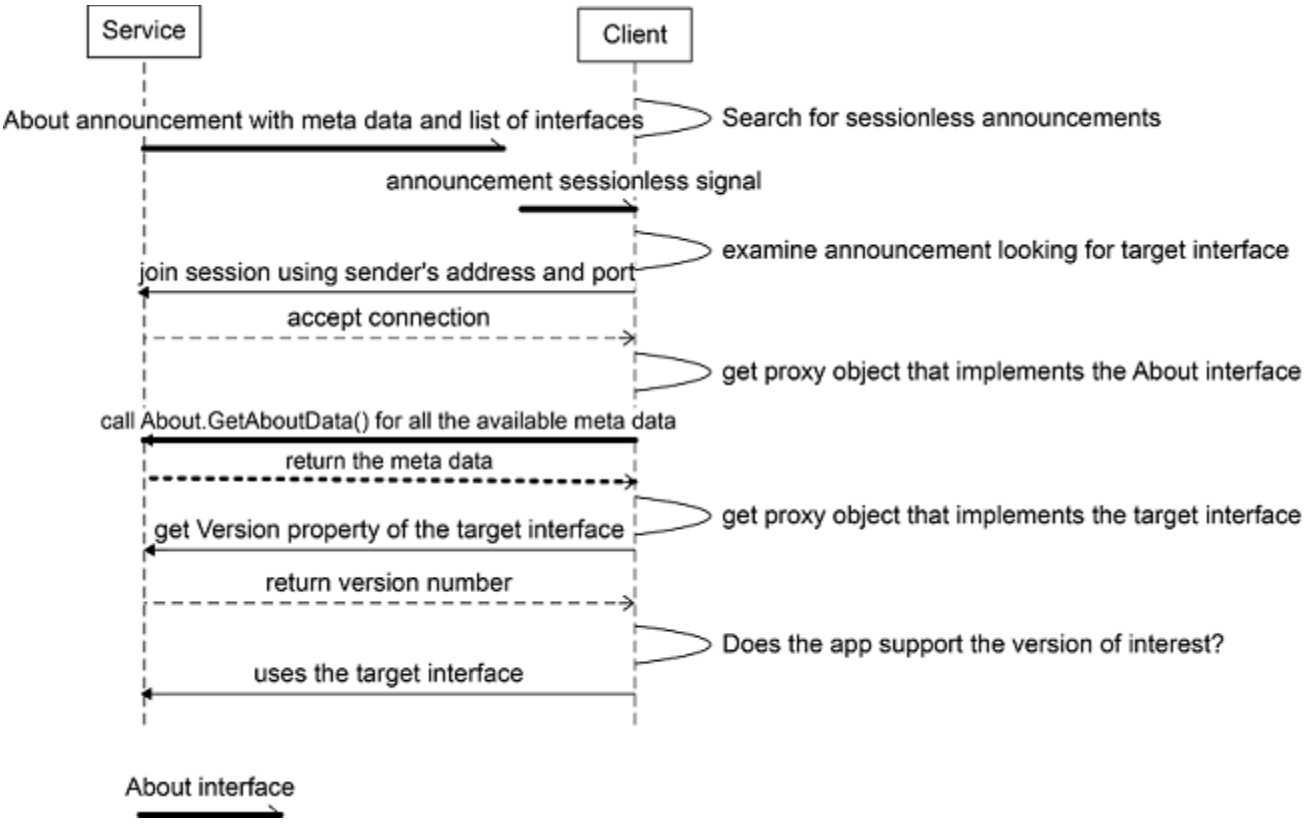


Figure: Nontypical discovery call flow

Error Handling

The method calls in the About interface will use the AllJoyn error message handling feature (ER_BUS_REPLY_IS_ERROR_MESSAGE) to set the error name and error message.

Error name	Error message
org.alljoyn.Error.LanguageNotSupported	The language specified is not supported

About Interface

Interface name	Version	Secured	Object path
`org.alljoyn.About`	1	no	`/About`

Properties

Property name	Signature	List of values	Read/Write	Description
Version	`q`	Positive integers	Read Only	Interface version number

Methods

The following methods are exposed by a BusObject that implements the `org.alljoyn.About` interface.

`a{sv} GetAboutData('s')`

Message arguments

Argument	Parameter name	Signature	List of values	Description
0	`languageTag`	`s`	IETF language tags specified by**RFC 5646**.	The desired language.

Reply arguments

Argument	Parameter name	Return signature	Description
0	`AboutData`	`a{sv}`	A dictionary of the available metadata fields. If language tag is not specified (i.e., ""), metadata fields based on default language are returned.

Error reply

Error	Description
`org.alljoyn.Error.LanguageNotSupported`	Returned if a language tag is not supported

Description

Retrieve the list of available AboutData fields based on the language tag. see **About data interface fields**

About data interface fields

The following table lists the names of the metadata fields. The fields with a yes value in the Announced column will also be published via the Announce signal. See **Signals** for more information.

Field name	Mandatory	Announced	Localized	Signature	Description
`AppId`	yes	yes	no	`ay`	A 128-bit globally unique identifier for the application. The AppId shall be a universally unique identifier as specified in**RFC 4122**.
`DefaultLanguage`	yes	yes	no	`s`	The default language supported by the device. Specified as an IETF language tag listed in**RFC 5646**.
`DeviceName`	no	yes	yes	`s`	Name of the device set by platform-specific means (such as Linux and Android).
`DeviceId`	yes	yes	no	`s`	Device identifier set by platform-specific means.
`AppName`	yes	yes	yes	`s`	Application name assigned by the app manufacturer

Field name	Mandatory	Announced	Localized	Signature	Description
					(developer or the OEM).
`Manufacturer`	yes	yes	yes	`s`	The manufacturer's name of the app.
`ModelNumber`	yes	yes	no	`s`	The app model number.
`SupportedLanguages`	yes	no	no	`as`	List of supported languages.
`Description`	yes	no	yes	`s`	Detailed description expressed in language tags as in RFC 5646.
`DateOfManufacture`	no	no	no	`s`	Date of manufacture using format YYYY-MM-DD (known as XML DateTime format).
`SoftwareVersion`	yes	no	no	`s`	Software version of the app.
`AJSoftwareVersion`	yes	no	no	`s`	Current version of the AllJoyn SDK used by the application.
`HardwareVersion`	no	no	no	`s`	Hardware version of the device on which the app is running.

Field name	Mandatory	Announced	Localized	Signature	Description
`SupportUrl`	no	no	no	`s`	Support URL (populated by the manufacturer).

`a(oas) GetObjectDescription()`

Message arguments

None.

Reply arguments

Argument	Parameter name	Return signature	Description
0	`objectDescription`	`a(oas)`	Return the array of object paths and the list of supported interfaces provided by each object.

Description

Retrieve the object paths and the list of all interfaces implemented by each of objects.

Signals

The following signals are emitted by a BusObject that implements the `org.alljoyn.About` interface.

`Announce('qqa(oas)a{sv}')`

Announce signal is a Sessionless signal

Message arguments

Argument	Parameter name	Signature	List of values	Description
0	`version`	`q`	positive	Version number of the About interface.
1	`port`	`q`	positive	Session port the app will listen on incoming sessions.

Argument	Parameter name	Signature	List of values	Description
2	`objectDescription`	`a(oas)`	Populated based on announced interfaces	Array of object paths and the list of supported interfaces provided by each object.
3	`aboutData`	`a{sv}`	array of key/value pairs	All the fields listed in **About data interface fields** with a yes value in the Announced column are provided in this signal.

AllJoyn Introspection XML

```
<node name="/About" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="http://www.allseenalliance.org/schemas/introspect.xsd"
>
   <interface name="org.alljoyn.About">
      <property name="Version" type="q" access="read"/>
      <method name="GetAboutData">
         <arg name="languageTag" type="s" direction="in"/>
         <arg name="aboutData" type="a{sv}" direction="out"/>
      </method>
      <method name="GetObjectDescription">
         <arg name="objectDescription" type="a(oas)" direction="out"/>
      </method>
      <signal name="Announce">
         <arg name="version" type="q"/>
         <arg name="port" type="q"/>
         <arg name="objectDescription" type="a(oas)"/>
         <arg name="metaData" type="a{sv}"/>
      </signal>
   </interface>
</node>
```

Icon Interface

Interface name	Version	Secured	Object path
`org.alljoyn.Icon`	1	no	`/About/DeviceIcon`

Properties

Property name	Signature	List of values	Read/Write	Description
`Version`	`q`	Positive integers	Read Only	Interface version number
`MimeType`	`s`	The Mime type corresponding to the icon's binary content	Read Only	Mime type for the icon
`Size`	`u`	The size in bytes of the icons binary content	Read Only	Size of the Icon

Methods

The following methods are exposed by a BusObject that implements the `org.alljoyn.Icon` interface.

`s GetUrl()`

Message arguments

None.

Reply arguments

Argument	Parameter name	Return signature	Description
0	`url`	`s`	The URL if the icon is hosted on the cloud

Description

Retrieve the URL of the icon if the icon is hosted on the cloud.

`ay GetContent()`

Argument	Parameter name	Return signature	Description
0	`content`	`ay`	The binary content for the icon

Signals

None.

AllJoyn Introspection XML

```
<node name="/About/DeviceIcon"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="http://www.alljoyn.org/schemas/introspect.xsd">
    <interface name="org.alljoyn.Icon">
        <property name="Version" type="q" access="read"/>
        <property name="MimeType" type="s" access="read"/>
        <property name="Size" type="u" access="read"/>
        <method name="GetUrl">
            <arg type="s" direction="out"/>
        </method>
        <method name="GetContent">
            <arg type="ay" direction="out"/>
        </method>
    </interface>
</node>
```

ABOUT API GUIDE

The About Feature was fully integrated with the core code for the AllJoyn® 14.12 release. Many of the API names were changed to fit more closely with naming used in the core code. The About Feature no longer needs a separate library. The About Feature can access more information. Making it possible to fill in some fields for the developer. If your application is still using the older APIs the legacy API guides can be used as reference.

- **Java**
- **C++**
- **Objective-C**
- **C (Thin Core)**

  Legacy API Guides
- **Java**
- **C++**

Common Best Practices

Handling BusListener::BusDisconnected

If you are writing an app intended for a platform that is running a standalone router (such as OpenWRT or Linux), it is recommended to register a Bus Listener and

implement `BusListener::BusDisconnected` to support scenarios when the connection between the AllJoyn® app and the AllJoyn router is lost.

This can happen in the following scenarios:

- The standalone router is restarted (manually or otherwise)
- The Onboarding service framework forces a restart of the standalone router after onboarding a device.

After `BusListener::BusDisconnected` is invoked:

- Clear now-obsolete application data such as session IDs.
- Shut down any service frameworks being used.
- Create a new bus attachment.
- Periodically invoke `BusAttachment::Connect` until it returns successfully.

After the AllJoyn router restarts and the new bus attachment is reconnected, any sessions the old bus attachment was previously a part of must be re-established to resume proper function. Likewise, any service frameworks must be restarted using the reconnected bus attachment.

The process is summarized below:

1. Verify the `BusListener` implements `BusListener::BusDisconnected`.
2. When `BusListener::BusDisconnected` is invoked, make sure to:

a. Clear any now-obsolete application data, such as session IDs.

b. Shut down any service frameworks being used.

c. Create a new bus attachment.

d. Continually try to reconnect the bus attachment.

e. Once reconnected:

i. Set listeners.

ii. Bind session ports as needed.

iii. Restart service frameworks.

iv. Connect to any pre-established sessions as needed.

Best Practices (across all services)

When to call the AboutService Announce() method

When using the About feature, the `Announce()` method should be invoked once all AllJoyn interfaces have been registered and whenever the data structure changes. A few scenarios for calling the `Announce()` method follow:

- Some embedded devices have certain functionality enabled through the device's Settings options. After any change to activate the AllJoyn interfaces, the `Announce()` method should be called again.
- Some embedded devices support configuring a name or other values that a user will enter. Each time there is a change, a call to `Announce()` should be made.

How the AboutProxy receives information

The `BusAttachment::RegisterAboutListener` registers for an AllJoyn Signal and does not need to poll, creating and then registering the `AboutListener` then calling the `BusAttachment::WhoImplements` method indicating the interfaces the client is interested in is all that is required. The `AboutListener` object should exist the lifetime of the application in order to receive up to date information. Once an announce signal is received by the `AboutListener::Announced` callback an `AboutProxy` object can be created to interact with the remote `AboutObj`.
For more information on `AboutListener`, refer to the About API Guide listed at the top of the page for the platform you are targeting.

Generating a unique AppId/unique ID

The About feature has a mandatory `AppId` field that requires a unique value be set per the application using.
This unique ID should follow the Internet Engineering Task Force (IETF) RFC 4122. This means the `AppId` will always be 128-bits in length. When setting the value, there is no need to use the "-" hyphen symbol; use the raw hex value and store it into a byte array.
The generation of the `AppId` can occur through various online offerings. Perform a search for "GUID generator" on various online search engines to aid in the generation of the `AppId`.

NOTE: If two or more applications use the same `AppId`, it does not hinder the `AboutObj` or its ability to interact with an `AboutProxy`. If an application using the `AboutProxy` relies on the `AppId` to display information, it may render incorrect results due to the non-unique `AppId`s.

When to send an Icon

Although not required, the About feature can support broadcasting and receiving an icon. The icon can be used by the applications to help visually identify the embedded device.

The recommended size for this icon is 72 x 72 pixels, but can be larger as long as the total number of bytes is less than the maximum supported by the AllJoyn framework in a single `BusMethod` call (`ALLJOYN_MAX_ARRAY_LEN`, 131072 bytes). If the icon image size is larger than`ALLJOY_MAX_ARRAY_LEN`, provide a valid URL when initializing the `AboutIconObj`.

In order for the icon to correctly render, it is important to set the mimeType to the image type as some devices require this to show the icon on a display.

Ping discovered devices

NOTE: The BusAttachment.Ping option is part of the AllSeen Alliance 14.06 release.

It is possible to receive an `org.alljoyn.About.Announce` signal with information about a bus name that is stale. Use the `BusAttachment::Ping` method to discover if the name is still present before trying to join a session with the remote bus.
A short timeout can be specified when calling the `BusAttachment::Ping` method. This can make applications more responsive since they will not have to wait as long for a `JoinSession` timeout failure.