Universitá di Pisa

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Magistrale in Computer Engineering

Tesi di Laurea Magistrale

Design and development of an AllJoyn to CoAP bridge

Candidato:
David Costa
Matricola 455356

Relatori:
Prof. Enzo Mingozzi
Ing. Carlo Vallati

Anno Accademico 2015–2016

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AJ AllJoyn

AJTCL AllJoyn Thin Core Library

API Application Programming Interface

CoRE Constrained RESTful Environments

CoAP Constrained Application Protocol

DSB Device System Bridge

DUT Device Under Test

GUID Globally Unique IDentifier

GUI Graphic User Interface

IoT Internet of Things

IP Internet Protocol

IPC Inter-Process Communication

M2M Machine to Machine

OS Operating System

RD Resource Directory

REST REpresentational State Transfer

RMI Remote Method Invocation

RPC Remote Procedure Call

SPQR Senatus PopulusQue Romanus

TCP Transmission Control Protocol

UDP User Datagram Protocol

URI Uniform Resource Identifier

WSN Wireless Sensor Network

XML eXtensible Markup Language

Abstract

The recent advancements of the building automation in the technological revolution of the Internet of Things (IoT) are leading to the need to enable the communication between devices extremely different from each other. Smart objects equipped with communication capabilities may rely on proprietary IoT solutions consisting of various hardware and software components, that would not be able to be interworked.

Several companies are developing advanced IoT software frameworks, e.g. AllJoyn, which enable interoperability between devices across multiple architectures and protocols. These frameworks provide resource discovery, data transmission and device management, and they support several platform and language bindings.

Despite their great effort, further improvements are still needed, due to technical limitations and framework availability of low-power devices in pre-existing networks. An example is the Constrained Application Protocol (CoAP), a redesign of the popular HTTP protocol that aims to support heavily resource-constrained devices for machine-to-machine (M2M) applications.

In this work we present the design and the implementation of an application that acts as a bridge between an AllJoyn network and a pre-existing one based on CoAP, in order to enable the resources provided by resource-constrained nodes to be reached by a more powerful network, which also includes computers and smartphones. The bridge allows on-demand registration of CoAP resources, dynamically translated and advertised on the AllJoyn network, so that AllJoyn client applications can easily discover and interact with them.

Experimental tests show the proper functioning of the bridge, the transparent way it operates, and the amplitude of its application. These tests validate the work with both an ad-hoc client application and an already existing AllJoyn application.

The first chapter shows an overview of the Internet of Things and the Smart Environment. There are the most famous frameworks disclosed, developed to enable interoperability among connected products and software applications across manufacturers. Then, we briefly introduce the CoAP protocol for resource-constrained devices and the idea of introducing the pre-existing network into the AllJoyn one.

The second chapter covers the whole AllJoyn framework. It includes its history, its main components and the system description which shows the framework features. This part also includes an overview of the D-Bus specification, on which AllJoyn is based.

In the third chapter we show the Device System Bridge. It is a bridge developed by Microsoft that enables the communication between AllJoyn and other networks (ZigBee, BACnet, Z-Wave). We explain the reason why we preferred to implement the CoAP bridge in another way, and what it would be like it if we had follow the Microsoft specification.

The forth chapter describes the data mapping process. In order to provide CoAP resources to the AllJoyn devices, these resources have to be represented as AllJoyn objects, as well as the CoAP messages. The chapter shows this translation process.

Chapter five contains the UML diagrams that model our system. Deployment, component and class diagrams show the bridge components on different abstraction levels. Sequence diagrams show the dynamic behaviour of the components in the system and how they interact with each other.

The last two chapter show the tests we have done. The sixth chapter is the test report, which includes the detailed information about the test setup and the obtained results, and it is achieved by using an ad-hoc AllJoyn application that acts as a client. The seventh chapter is a demonstration of the operation of our bridge in the Windows environment, using real temperature and light sensors, and the bridge running on a Linux server. Since Windows 10 supports AllJoyn natively, we used a Microsoft application as test client.

1

INTRODUCTION

In this first chapter we introduce the Internet of Things with particular focus on the Smart Environment, like the Smart Home and, more in general, the Smart Building. The Smart Environment includes the proximity services as well as the resource-constrained devices, and it also focuses on the communication between different technologies.

Here we present some of the most important frameworks that enable the Smart Environment and that act as a glue between different devices and platforms. Our attention focuses on the AllJoyn framework, that seems to be the leader in its space.

With regard to the resource-constrained devices, one of the most widely used protocols is the Constrained Application Protocol (CoAP), a standardized software protocol intended to be used in very simple electronics devices.

Then, a problem of interoperability arises between more powerful devices based on the new frameworks and the pre-existing networks with very resource-constrained devices.

1.1 INTERNET OF THINGS

1.1.1 *State of the art*

The Internet of Things (IoT) is a computing concept that describes a network of physical objects embedded with electronics, software, sensors, and network connectivity throw which these objects collect and exchange data. Physical objects adopt an IP address for internet connectivity, so that the IoT extends the internet network beyond traditional devices like desktop and laptop computers, smartphones and tablets to a diverse range of devices.

IoT is becoming an increasing topic of interest among technology giants and business communities. Estimates set the current number of connected devices at about 6.4 billion, and industry expectations say that the tally will increase to 20.8 billion by 2020 [1].

The Internet of Things enables the implementation of technologies such as smart grids, smart homes, intelligent transportation and smart cities. In particular, the building automation has become one of the major points of interest of recent years.

The Smart Building, or its residential extension, the Smart House, is used to monitor and control the mechanical, electrical and electronic systems inside a building, e.g., at home, in the hospital, or in the factory. It covers heating, ventilation and air conditioning

(HVAC), lighting control system, security, automation for the elderly and disabled, and all sort of other applications.

The Smart Building is mainly implemented in two wireless ways:

- Machine to Machine communication in wireless personal area network (WPAN). IEEE standard 802.15.4 often offers the fundamental lower network layers for this kind of network, which focuses on low-power devices. Here, the IoT devices nearby each other will form proximal IoT networks.
- Powerful communication in wireless local area network (WLAN) based on the IEEE 802.11 standards. Devices which use Wi-Fi technology include personal computers, smartphones and tablets.

1.1.2 *Challenges and limitations*

Important features for an IoT system are the network ability to change over time and the device ability to discover resources, so that smart devices within each IoT proximal network can dynamically join and leave the network, and discover and communicate with each other. These key design aspects allow the implementation of IoT use cases in non-static environments, e.g., hotels and hospitals, where the involved nodes may continuously change, or scenarios in which always-on device availability is not required. The discovery should happen automatically based on proximity criteria and application requirements.

Then, an 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 proximal network, and consumer applications interested in these services discover them. Consumer applications then connect to provider applications to make use of these services as desired.

One of the remaining critical points in the IoT is the interoperability between devices across multiple architectures, platforms and language bindings. The proliferation of communication protocols and data formats across the device ecosystem makes it difficult for nodes to speak the same language and work together in harmony. In order to realize the full potential of IoT, technologies which enable devices and applications to interact each other are needed.

Several companies are working for that common language: software frameworks that serve as glue to allow devices from different companies, running on different operating systems, written with different language bindings to all speak together, and just work.

1.2 IMPLEMENTATION OF IOT SYSTEMS

1.2.1 *Software frameworks*

Advanced software frameworks, which abstract and isolate the developer from the complexity of the hardware and the networking sub-systems, re-define the development and re-usability of integrated hardware and software solutions. These frameworks enable interoperability among connected products and software applications, across manufacturers, to create dynamic proximal networks.

In recent years, several consortiums are developing their own IoT frameworks with focus on the Smart Environment. Industry consortiums are groups of companies which are working together to produce common standards for the benefit of themselves and the industry. The consortiums that have made greater effort in this area are the OIC (Open Interconnect Consortium), which oversees all aspects of *IoTivity*, and AllSeen Alliance, which is responsible of the development, marketing and certification around the *AllJoyn* platform.

The IoTivity is an open source project. It is developed on the constrained application protocol (CoAP), which means it is based on a resource-based, RESTful architecture model.

AllJoyn is an open source project too. It is based on the D-Bus specification, a remote procedure call (RPC) mechanism, extended by AllJoyn to work on remote devices.

From an OSI network model perspective, IoTivity and AllJoyn are full-stack providers. If you build a device to work with these, the entire protocol all the way to the application layer is specified. Even if the architecture of the two frameworks is quite different, they work with the same goal of creating a new standard by which billions of wired and wireless devices will connect to each other and to the Internet. Both of them shall provide resource discovery, data transmission and device management, and both implement a tiny version of the system for constrained devices.

In order to enable interoperability, these two software frameworks support several platforms and bindings:

- IoTivity supports Android, Arduino, Linux, Tizen and Yocto. Its core functionality is written in C, and other available bindings are C++ and Java.
- AllJoyn supports Android, Arduino, iOS, Linux, OS X, Windows and OpenWRT. Furthermore, Windows 10 includes the AllJoyn framework as core component. Its core is written in C++ and AllJoyn provides C, Java, JavaScript, C# and Obj-C bindings.

What has been said shows the great flexibility of AllJoyn regarding multi-platform interoperability. Moreover, the AllJoyn community is far larger than the IoTivity one, and there is a wider range of devices supporting it already available in the market.

That, coupled with the absence of cooperation with CoAP, led us to opt for a work on the AllJoyn framework.

1.2.2 *Constrained Application Protocol*

The *Constrained Application Protocol* (CoAP) is a specialized web transfer protocol to be used with constrained nodes and constrained networks in the Internet of Things. The nodes often have 8-bit microcontrollers with small amounts of ROM and RAM, while constrained networks such as IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs) often have high packet error rates and a typical throughput of 10s of kbit/s.

Like the Hypertext Transfer Protocol (HTTP), CoAP is based on the wildly successful REST model. This means that servers make resources available under a URL, and clients access these resources using methods such as GET, PUT, POST, and DELETE.

CoAP is designed to use minimal resources, both on the device and on the network. Instead of a complex transport stack, it gets by with UDP on IP. A 4-byte fixed header and a compact encoding of options enables small messages that cause no or little fragmentation on the link layer. CoAP makes use of two message types, requests and responses.

CoAP has the following main features:

- Web protocol fulfilling M2M requirements in constrained environments.
- UDP binding with optional reliability supporting unicast and multicast requests.
- Asynchronous message exchanges.
- Low header overhead and parsing complexity.
- URI and Content-type support.
- Simple proxy and caching capabilities.

The Internet Engineering Task Force (IETF) Constrained RESTful environments (CoRE) Working Group has done the major standardization work for this protocol [2].

Since it is one of the most widely used protocols in M2M and IoT, along with MQTT and LWM2M, and since its success has been well-established over the years, we focus on CoAP.

1.2.3 *Heterogeneous networks communication*

One of the greatest limitations of these IoT frameworks is the absence of interoperability with pre-existing networks. This limitation reflects into three main problems:

- Even if the systems provide a tinier version of themselves for constrained devices, not all the low-power devices have enough system resources to support the thin libraries. Sometimes it may be possible to get the tiny version onto these platforms but they will then have no memory left for the application logic.
- Some networks may use communication technologies that are not supported by the frameworks. For example, CoAP is based on the IEEE 802.15.4 standard that is not yet included in the AllJoyn system.
- Pre-existing network means that the network was present before the new network was installed. If we are making a Smart Building project, it would be convenient to allow cooperation between them, rather than replace the old one.

The interaction between heterogeneous networks should be done in a transparent manner, so that an application can communicate with other-network devices as if those devices were based on the same technology.

It would be even better if integration of different devices was realized on-demand. This would mean making CoAP resources available to the AllJoyn applications on request of the CoAP servers, and without pre-configurations.

The aim of the work is to enable the communication between the two different networks we work on, making CoAP devices available for an AllJoyn network, and then to allowing AllJoyn applications to interact with CoAP resources. This means that a pre-existing network, composed by heavily resource-constrained devices, could be reached by a more powerful network, which also includes computers and smartphones. Moreover, it is done without the need of a previous configuration of each device.

2

ALLJOYN

The software framework on which we focus our attention is AllJoyn. In this chapter we present it, the reason why it was developed and a system overview in which its functionalities are shown.

Then, the D-Bus is briefly described, on which AllJoyn is based. The framework uses the D-Bus specification in the naming guidelines, in its data type and in the messages format.

Finally, we analyse each component of the AllJoyn framework with a system description. In order to cover the whole system, a brief outline is given for each of those component. The description shows the basic concepts of AllJoyn, like the sessions and what is meant by apps and routers, and it includes an explanation on how the framework behaves during advertisement, discovery and data exchange phases.

2.1 SYSTEM OVERVIEW

2.1.1 *What is AllJoyn?*

AllJoyn is a device discovery and control software framework that was created by Qualcomm and then open-sourced. Now developed under the guide of the AllSeen Alliance, itself governed by the Linux Foundation, AllJoyn seems to be the leader in its space.

The AllSeen Alliance is a cross-industry consortium dedicated to driving the widespread adoption of products, systems and services that support the Internet of Everything. The Alliance hosts and advances an industry-supported open software connectivity and services framework based on the AllJoyn open source project.

The AllSeen Alliance has been created to promote some type of interoperability for the Internet of Things, and a number of consumer brands have signed on including Microsoft, LG, Sharp, Canon, Haier, Sony, Electrolux, Qualcomm and Arçelik. Other members include Affinergy, Asus, Canary, Cisco, Hisilicon, HTC, IBM, LIFX, Muzzley, Panasonic, TP-Link, Vodafone, and ZTE [3].

The AllSeen Alliance gives the following description to the AllJoyn framework [4]:

> *AllJoyn is an open source software framework that makes it easy for devices and apps to discover and communicate with each other. Developers can write applications for interoperability regardless of transport layer, manufacturer, and without the need for Internet access.*

That is because 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.

One of the defining traits of the AllJoyn framework is its inherent flexibility. It was designed to run on multiple platforms and 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)

In order to fully realize the vision of the Internet of Things, devices and applications need a common way to interact and speak to each other. The AllJoyn framework can serve as the glue to allow devices from different companies, running on different operating systems, written with different language bindings to all speak together.

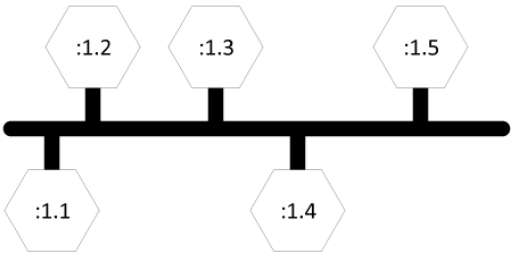
2.1.2 *Conceptual overview*

The most basic abstraction of the AllJoyn system is the *AllJoyn bus* (Figure 1a). It provides a fast, lightweight way to move marshaled messages around the distributed system. The AllJoyn application connects to a local AllJoyn bus and each connection is assigned a unique connection name.

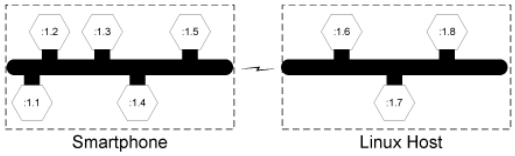
The AllJoyn bus is typically extended across devices as shown in Figure 1b. The figure illustrates that the logical distributed bus is actually split up into a number of segments, each running on a different device. The AllJoyn functionality that implements these logical bus segments is called an *AllJoyn router*.

These routing nodes coordinate the message flow across the logical bus, which appears as a single entity to the connections. In this configuration, client component can make remote method calls to service component as if it were a local object.

That is a *Remote Method Invocation*: a program calls a procedure located in another address space on a physically separate machine. RMI provides a proxy that implements an interface which looks just like that of the remote object. A proxy object is a local representation of a remote object that is accessed through the bus.



(a) Prototypical AllJoyn bus



(b) A distributed AllJoyn bus

Figure 1: AllJoyn Bus

Therefore, the AllJoyn framework is fundamentally an object-oriented system. 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.

Since the devices have been physically separated, there is no way for the involved bus routers to have any knowledge of the others. In order to determine that the other routers exist, and to determine that there is any need to connect to each other and form a logical distributed AllJoyn bus, AllJoyn provides advertisement and discovery. 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.

2.1.3 *Software architecture*

The AllJoyn network comprises *AllJoyn Applications* and *AllJoyn Routers*. Figure 2 shows the AllJoyn software architecture.

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.

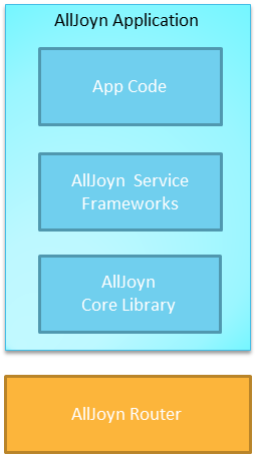


Figure 2: AllJoyn Software Architecture

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 definition of methods, properties, and signals
- Object creation and handling

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, applications and devices can properly interoperate with each other to perform a specific functionality.

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.
- Thin, for resource-constrained embedded devices with limited memory.

Components of the *AllJoyn Standard Core Library* are designed to run on Microsoft Windows, Linux, Android, iOS, OS X, and OpenWRT. 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 and user interfaces. *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, having many network connections, and using relatively large amounts of RAM and ROM.

D-Bus

AllJoyn provides its own bus based on D-Bus Wire protocol, described in Section 2.2. The AllJoyn system makes use of the D-Bus specification as follow:

- 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.
- It uses D-Bus naming guidelines for naming well-known names, interface names and object path names.
- It uses a D-Bus defined Simple Authentication and Security Layer framework for application layer authentication between applications.

2.2 D-BUS SPECIFICATION

D-Bus is an inter-process communication (IPC) and remote procedure call (RPC) mechanism that allows communication between multiple processes concurrently running on the same machine. D-Bus has the merit of being low-overhead because it uses a binary protocol, and does not have to convert to and from a text format such as XML.

The AllJoyn system re-implements the wire protocol set forth by the D-Bus specification and extends the D-Bus wire protocol to support distributed devices. In this section are described they key points of the D-Bus specification [5].

2.2.1 *Type system*

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

The D-Bus protocol does not include type tags in the marshaled data; a block of marshaled values must have a known type signature. The type signature is made up of zero or more single complete types, each made up of one or more type codes. A type code is an ASCII character representing the type of a value.

The simplest type codes are the *basic types*, which are the types whose structure is entirely defined by their 1-character type code. Basic types consist of fixed types and string-like types.

The *fixed types* are basic types whose values have a fixed length, namely *BYTE*, *BOOLEAN*, *DOUBLE*, *UNIX_FD*, and signed or unsigned integers of length 16, 32 or 64 bits. The characteristics of the fixed types are listed in Table 1.

NAME	ASCII TYPE-CODE	ENCODING
BYTE	y	Unsigned 8-bit integer
BOOLEAN	b	Boolean value
INT16	n	Signed (two's complement) 16-bit integer
UINT16	q	Unsigned 16-bit integer
INT32	i	Signed (two's complement) 32-bit integer
UINT32	u	Unsigned 32-bit integer
INT64	x	Signed (two's complement) 64-bit integer
UINT64	t	Unsigned 64-bit integer
DOUBLE	d	Double-precision floating point

Table 1: D-Bus Fixed Types

The *string-like types* are basic types with a variable length. The value of any string-like type is conceptually *0* or more Unicode codepoints encoded in UTF-8. The characteristics of the string-like types are listed in Table 2.

NAME	ASCII TYPE-CODE
STRING	s
OBJECT PATH	o
SIGNATURE	g

Table 2: D-Bus String-like Types

In addition to basic types, there are some *container types*, among which stand out the *STRUCT* and the *ARRAY*.

STRUCT has a type code, ASCII character *'r'*, but usually this type code does not appear in signatures. Instead, ASCII characters *'('* and *')'* are used to mark the beginning and end of the struct.

ARRAY has ASCII character *'a'* as type code. The array type code must be followed by a single complete type and the single complete type following the array is the type of each array element.

The container types include also the *VARIANT* (ASCII character *'v'*). A marshaled value of type *VARIANT* will have the signature of a single complete type as part of the value. This signature will be followed by a marshaled value of that type.

2.2.2 *Message format*

A message consists of a header and a body. The header is a block of values with a fixed signature and meaning. The body is a separate block of values, with a signature specified in the header.

The signature of the header is:

$$"yyyyuua(yv)"$$

These values have the meanings described in Table 3.

VALUE	DESCRIPTION
1st BYTE	Endianness flag
2nd BYTE	Message type
3rd BYTE	Bitwise OR of flags
4th BYTE	Major protocol version of the sending application
1st UINT32	Length in bytes of the message body, starting from the end of the header
2nd UINT32	The serial of this message
ARRAY of (BYTE,VARIANT)	An array of zero or more header fields where the byte is the field code, and the variant is the field value

Table 3: D-Bus Message Header

Message types that can appear in the second byte of the header are: *INVALID*, *METHOD_CALL*, *METHOD_RETURN*, *ERROR*, *SIGNAL*.

The array at the end of the header contains header fields, where each field is a 1-byte field code followed by a field value. The message type determines which fields are required. The header fields include the object path, the interface, the member name, the sender and the destination, among other fields.

2.2.3 *Valid names*

The various names in D-Bus messages have some restrictions. There is a maximum name length of 255 which applies to bus names, interfaces, and members.

Interface names

Interfaces have names with type *STRING*, meaning that they must be valid UTF-8. However, there are also some additional restrictions that apply to interface names specifically:

- Interface names are composed of 1 or more elements separated by a period (*'.'*) character. All elements must contain at least one character.
- Each element must only contain the ASCII characters *"[A-Z][a-z][0-9]_"* and must not begin with a digit.
- Interface names must contain at least one *'.'* character (and thus at least two elements).
- Interface names must not begin with a *'.'* character.
- Interface names must not exceed the maximum name length.

Interface names should start with the reversed DNS domain name of the author of the interface (in lower-case), like interface names in Java.

For instance, if the owner of *example.com* is developing a D-Bus API for a music player, they might define interfaces called *com.example.MusicPlayer*.

Bus names

Connections have one or more bus names associated with them. A connection has exactly one bus name that is a unique connection name. The unique connection name remains with the connection for its entire lifetime. A bus name is of type STRING, meaning that it must be valid UTF-8. However, there are also some additional restrictions that apply to bus names specifically:

- Bus names that start with a colon (*':'*) character are unique connection names. Other bus names are called well-known bus names.
- Bus names are composed of 1 or more elements separated by a period (*'.'*) character. All elements must contain at least one character.
- Each element must only contain the ASCII characters *"[A-Z][a-z][0-9]_-"*. Only elements that are part of a unique connection name may begin with a digit, elements in other bus names must not begin with a digit.
- Bus names must contain at least one *'.'* character (and thus at least two elements).

- Bus names must not begin with a '.' character.
- Bus names must not exceed the maximum name length.

Like interface names, well-known bus names should start with the reversed DNS domain name of the author of the interface (in lower-case).

For instance, if the owner of *example.com* is developing a D-Bus API for a music player, they might define that any application that takes the well-known name *com.example.MusicPlayer* should have an object at the object path */com/example/MusicPlayer* which implements the interface *com.example.MusicPlayer*.

Member names

Member (i.e. method or signal) names:

- Must only contain the ASCII characters *"[A-Z][a-z][0-9]_"* and may not begin with a digit.
- Must not contain the '.' character.
- Must not exceed the maximum name length.
- Must be at least 1 byte in length.

It is conventional for member names on D-Bus to consist of capitalized words with no punctuation ("camel-case"). Method names should usually be verbs, such as *GetItems*, and signal names should usually be a description of an event, such as *ItemsChanged*.

2.2.4 *Message types*

Each of the message types (*METHOD_CALL*, *METHOD_RETURN*, *ERROR*, and *SIGNAL*) has its own expected usage conventions and header fields.

Method calls

Some messages invoke an operation on a remote object. These are called method call messages and have the type tag *METHOD_CALL*. Such messages map naturally to methods on objects in a typical program.

A method call message is required to have a *MEMBER* header field indicating the name of the method. Optionally, the message has an *INTERFACE* field giving the interface the method is a part of.

Method call messages also include a *PATH* field indicating the object to invoke the method on. If the call is passing through a message bus, the message will also have a *DESTINATION* field giving the name of the connection to receive the message.

When an application handles a method call message, it is required to return a reply. The reply can have one of two types, either *METHOD_RETURN* or *ERROR*. If the reply has type *METHOD_RETURN*, the arguments to the reply message are the return value.

Signal emission

Unlike method calls, signal emissions have no replies. A signal emission is simply a single message of type *SIGNAL*. It must have three header fields: *PATH* giving the object the signal was emitted from, plus *INTERFACE* and *MEMBER* giving the fully-qualified name of the signal. The *INTERFACE* header is required for signals, though it is optional for method calls.

Errors

Messages of type *ERROR* are most commonly replies to a *METHOD_CALL*, but may be returned in reply to any kind of message. The message bus for example will return an *ERROR* in reply to a signal emission if the bus does not have enough memory to send the signal.

An *ERROR* may have any arguments, but if the first argument is a string, it must be an error message.

2.3 SYSTEM DESCRIPTION

In this section is described how the AllJoyn framework works, which includes its main components and how they are composed, the discovery of services, the interaction between applications, and other features. The system description follows the AllJoyn documentation provided by the AllSeen Alliance [6].

2.3.1 *AllJoyn system key concept*

AllJoyn router

The *AllJoyn Router* component provides core functionality of the AllJoyn system, including peer-to-peer advertisement and discovery, connection establishment, broadcast signaling and 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.

An AllJoyn router can live on the same physical device of the application, or on different devices. Three common topologies exist, shown in Figure 3

1. An application uses its own Router. In this case, the Router is called a *"Bundled Router"* as it is bundled with the application. Mobile applications usually fall in this group.

2. Multiple applications on the same device use one Router. In this case, the Router is called a *"Standalone Router"* and it typically runs in a background process. This is common on Linux systems where the AllJoyn Router runs as a daemon process.

3. An application uses a Router on a different device. Embedded devices, which use the AJTCL, typically fall in this camp.

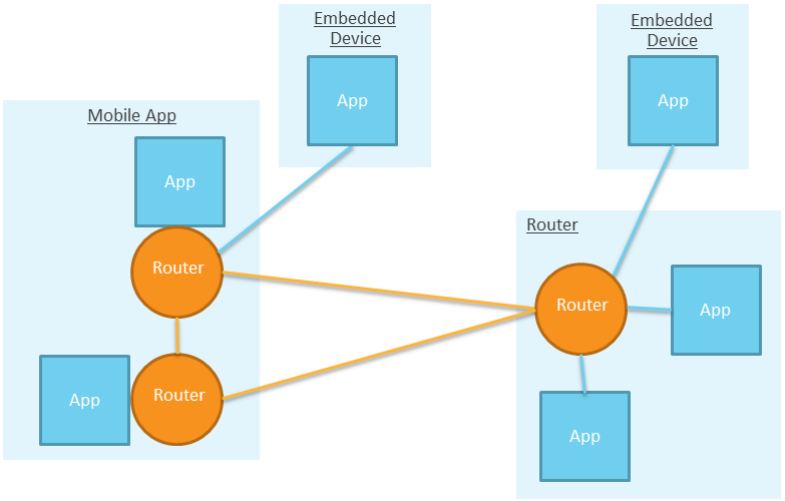


Figure 3: AllJoyn Router

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. The logical AllJoyn bus maps to a single AllJoyn router in the case of standalone deployment model or bundled deployment model with only one application on the device. Instead, the logical AllJoyn bus maps to multiple AllJoyn router instances in the bundled deployment model with multiple applications on the device.

Multiple instances of AllJoyn buses across multiple devices form a logical distributed AllJoyn software bus. The distributed AllJoyn bus hides all the communication link details from the applications running on multiple devices.

Unique name

In order 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 following format:

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

Well-known name

A well-known name is a consistent way to refer to a service offered over the AllJoyn bus. 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. However, its uniqueness is guaranteed only within the local AllJoyn bus.

In order to distinguish multiple instances of a given application on the AllJoyn bus, the well-known name should have a unique identifier as a suffix The AllJoyn well-known name follows the D-Bus specification guidelines for naming and has following format:

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

AllJoyn object

AllJoyn applications implement one or more AllJoyn objects to support AllJoyn services functionality. These AllJoyn objects 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, which is a local representation of a remote service object that is accessed through the AllJoyn bus.

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, and 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.

The object path naming also adheres to the D-Bus specification naming guidelines:

Object Path = /Application/Object

AllJoyn interface

Each AllJoyn object exposes its functionality over the AllJoyn bus through one or more AllJoyn interfaces. An AllJoyn interface can include one or more of following types of members:

METHODS A method is a function call that performs some processing and typically returns outputs reflecting the results of the processing operation.

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.

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. Similar to the well-known name, the AllJoyn interface name also follows reverse domain name format and D-Bus specification naming guidelines:

Interface Name = com.example.Interface

2.3.2 *Advertisement and discovery*

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

Applications can use one of the following methods to advertise and discover services over the AllJoyn framework:

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

Name-based discovery

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

The Name Service is implemented using *IS-AT* and *WHO-HAS* protocol messages, which carry well-known names to be advertised and discovered, respectively. These protocol messages are multicast over the AllJoyn proximal network over IANA-registered IP multicast groups and port number as listed in Table 4.

TYPE	ADDRESS
IPv4 Multicast group address	224.0.0.113
IPv6 Multicast group address	FF0X::13A
Multicast port number	9956

Table 4: IANA multicast address

The IS-AT message advertises AllJoyn services using the well-known name or the unique name. A single IS-AT message can include a list of one or more well-known names or unique names for advertisement. The AllJoyn router at the provider device send out IS-AT message periodically over IP multicast to advertise the set of services it supports. The IS-AT message can also be sent out in response to a received WHO-HAS message that is looking for that advertised service.

The WHO-HAS message discovers one or more AllJoyn services using the well-known name or the unique name. Similar to IS-AT, a WHO-HAS message can include a list of one or more well-known names or unique names for discovery. When a consumer device wants to discover a service, it sends out the WHO-HAS message over IP multicast.

Announcement-based discovery

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

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

2.3.3 *AllJoyn transport*

The AllJoyn Transport supports connections establishment and delivering messages over multiple underlying physical transport layers including TCP, UDP and local UNIX transport. AllJoyn Transport functionality can be divided into two categories:

- *Local AllJoyn Transports*: designed to essentially provide communication between Core Library and associated AllJoyn Router.
- *Bus-to-Bus AllJoyn Transports*: enable connection establishment and message routing between AllJoyn routers.

Although the primary task of an AllJoyn transport is to transport, or move, AllJoyn messages from one endpoint to another, it is important to distinguish the AllJoyn Transport from the concept of transport layer (layer 4) in the ISO/OSI 7-layer model, as shown in Figure 4.

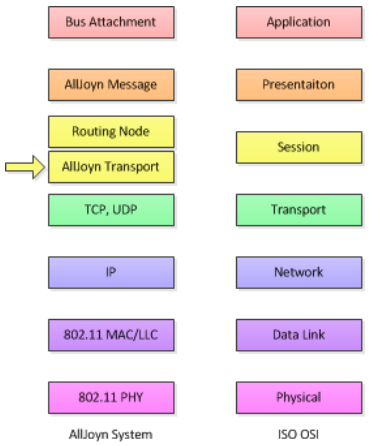


Figure 4: AllJoyn in the ISO/OSI 7-layer model

Underneath the Application Logic, there exists an AllJoyn Message layer which is responsible for marshaling and unmarshaling of AllJoyn messages (Signals and Method Calls). This layer can be thought of as residing in the presentation layer (layer 6) of the ISO/OSI model.

These AllJoyn messages are routed to their intended destination by the AllJoyn Transport layer. Since the AllJoyn Transport layer manages connections across applications and AllJoyn routers in the network, it can be thought of corresponding to the session layer (layer 5) of the ISO/OSI model. AllJoyn Transports make use of layer 4 transports like TCP or UDP in order to manage the actual movement of AllJoyn messages between various network entities.

Bus-to-Bus AllJoyn Transports

The Bus-to-Bus AllJoyn Transports enable connection establishment and message routing between AllJoyn routers. The most commonly used Bus-to-Bus transports in the AllJoyn system are based on the underlying IP-based transport mechanisms. These include TCP Transport and UDP Transport. An AllJoyn applications may select the AllJoyn Transport that is actually used by choosing one or more *TransportMask* bits in the appropriate AllJoyn APIs.

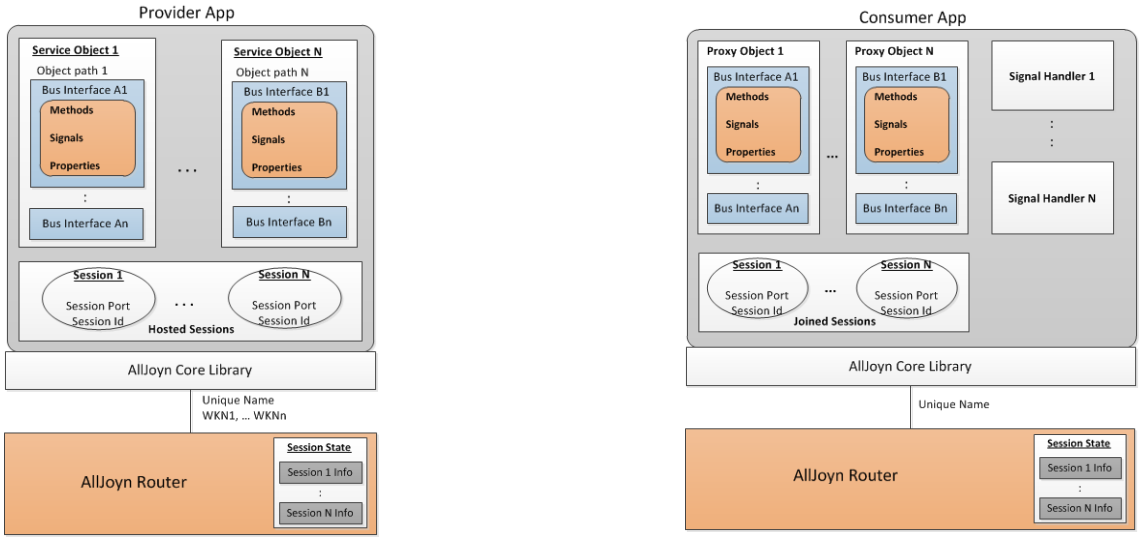
Since TCP provides a reliable data stream guarantee, the TCP Transport must only provide enough mechanism to translate AllJoyn messages to and from byte streams.

Since UDP does not provide a reliability guarantee, the UDP Transport must provide some mechanism to provide a reliable message delivery guarantee. The UDP Transport uses the *AllJoyn Reliable Datagram Protocol* (ARDP) to provide reliable delivery of messages. ARDP is based loosely on the *Reliable Data Protocol* (RDP) [7].

2.3.4 *Data exchange*

The AllJoyn provider application (Figure 5a) implements one or more service objects that provide service functionality. These service objects implement one or more Bus interfaces which support methods, signals, and properties as interface members. AllJoyn applications can exchange data using these interface members. An AllJoyn session, described in Section 2.3.5, must be established to exchange data between provider and consumer applications except when sending sessionless signals.

After an AllJoyn session is established, the consumer application (Figure 5b) can invoke methods and properties on the remote service objects, or can opt to receive signals emitted by the provider application. A *ProxyBusObject* is needed to exchange data via methods and properties, and a *signal handler* is needed to receive signal data from the provider application.



(a) Provider application

(b) Consumer application

Figure 5: Functional architecture

Match rules

The AllJoyn framework supports D-Bus match rules for the consumer application to request and receive specific set of messages. Match rules describe the messages that should be sent to a consumer application based on the contents of the message. Consumer applications can add a match rule by using the *AddMatch* method exposed by the AllJoyn router. Match rules are specified as a string of comma-separated key/value pairs.

For example:

Match Rule = "type='signal',interface='org.freedesktop.DBus',path='/bar/foo'"

Message routing

The AllJoyn framework uses the D-Bus marshaling format and the D-Bus message format (Section 2.2) and extends it with additional header flags and header fields for AllJoyn messages. The AllJoyn message format is used to send messages between AllJoyn routers as well as between the application and the AllJoyn router.

The AllJoyn system supports message routing based on session ID and destination fields for application-specific messages. A session ID-based routing table is formed and maintained at the AllJoyn router for routing messages. Conceptually, for every session ID, the routing table maintains a list of destination application endpoints for every application participating in the session and next hop bus-to-bus endpoint for those application endpoints which are remote.

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

2.3.5 *AllJoyn session*

After an AllJoyn consumer has discovered some desired services offered by provider devices, the next step is to establish an AllJoyn session with the provider to consume those services. An AllJoyn session is a logical connection between consumer and provider applications that allows these applications to communicate with each other and exchange data.

A provider application creates an AllJoyn session and waits for consumer applications to join the session. The provider AllJoyn router assigns a unique session ID for the session and also creates a session map storing the relevant session information.

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

POINT-TO-POINT SESSION An AllJoyn session with a single consumer (joiner) and single provider (session host) endpoints participates in the session.

MULTI-POINT SESSION An AllJoyn session that involves a provider application (session host) and one or more consumer applications (joiners) participating in the same session.

A compatible set of session options must be agreed upon between two endpoints to establish a session. If a compatible set of session options cannot be established between two endpoints, session establishment fails. Table 5 capture the session options supported for the AllJoyn session.

OPTION	DESCRIPTION
traffic	Specifies if type of traffic sent over the session is message-based or raw.
isMultipoint	Specifies whether the session is multi-point or point-to-point.
proximity	Specifies the proximity scope for this session (physical or network).
transport	Specifies the allowed transports for this Session between TCP, UDP or any.

Table 5: Session options

2.3.6 *Sessionless signal*

The sessionless signal is an AllJoyn feature that enables broadcasting of signals to all reachable nodes in the AllJoyn proximal network. This is different than the session-based signals described in Section 2.3.5, where signals are sent only to participants connected over a given session or multiple sessions.

Sessionless signals are logically broadcast signals and any application on the AllJoyn proximal network interested in receiving sessionless signals will receive all sessionless

signals sent by any other application on that network. The AllJoyn system design refers to sessionless signals as logically broadcast because signals themselves are not broadcast, only an indication for signals is sent over multicast to all the nodes on the network.

Applications can specify match rules, as described in Section 2.3.4, to receive a specific set of sessionless signals and the AllJoyn router filters out signals based on those match rules.

2.3.7 *AllJoyn security*

The AllJoyn system provides a security framework for applications to authenticate each other and send encrypted data between them. Authentication and data encryption are done at the application level.

An application can tag an interface as secure to enable authentication and encryption. Then, all of the methods, signals, and properties of a secure interface are considered secure. Authentication and encryption related key exchange are initiated on demand when a consumer application invokes a method call on a secure interface, or explicitly invokes an API to secure the connection with a remote peer application.

Authentication and encryption keys are stored in a *Key Store*. Storage and data used to implement the AllJoyn security are described below.

Key store

The *Key Store* is a local storage used to persistently store authentication-related keys, and to store master secret and associated TTL. Multiple applications on a device can share a given key store.

Authentication GUID

The *Authentication GUID* is a GUID assigned to an application for authentication purposes. This GUID is persisted in the key store and provides a long-term identity for the application.

Master secret

The *Master Secret* is a key (48 bytes long) shared between authenticated peer applications. Two peer applications generate the same master secret independently, and store it persistently in the key store.

Session key

A cryptographic *Session Key* (128 bits long) used to encrypt point-to- point data traffic between two peer applications. A separate session key is maintained for every connected peer application. A session key is valid as long as peers are connected.

Group key

The *Group Key* (128 bits long) is a cryptographic key used to encrypt point-to-multipoint data traffic sent out by a provider application. Only provider applications use the group key to send out encrypted broadcast signals. Applications exchange their group keys using an encrypted method call that involves the session key.

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

The session key and the group keys are generated using the algorithm described in the Transport Layer Security Protocol [8]. Message encryption is done using AES CCM algorithm [9].

3

DEVICE SYSTEM BRIDGE

In order to realize the full capabilities of AllJoyn, companies and users might need to replace their existing device with devices that support AllJoyn or update their existing devices with AllJoyn-capable firmware. In many cases, these options are not feasible due to device cost, technical limitations, or AllJoyn firmware availability for the device.
In order to address these challenges, an element that behaves as a bridge between AllJoyn and other technologies is needed.

Microsoft developed its own bridge and called it *Device System Bridge* [10]. The DSB enables non-AllJoyn devices to be included in the AllJoyn ecosystem. A DSB uses existing device interfaces to access the non-AllJoyn devices and creates a virtual proxy of these devices on the AllJoyn bus.

In this chapter we describe the DSB made by Microsoft, the reason why we preferred to implement it in another way, and what it would be like our bridge if we had follow the Microsoft specification.

3.1 MICROSOFT DSB

Each DSB is implemented as a separate AllJoyn application. Windows 10 provides an AllJoyn router node as part of the OS and the DSB depends on these OS supplied AllJoyn APIs and the AllJoyn Framework for enablement. In Windows, the DSB is designed as a single Universal Application for each DSB type, e.g. Z-Wave or BACnet.

3.1.1 *DSB overview*

A DSB has three structural components:

COMMUNICATION STACK provides interconnection to the purpose built device system.

ADAPTER instantiates and manages a virtual device on behalf of each device from the alternative network that can be exposed to the AllJoyn bus. This information is consumed by the Bridge.

BRIDGE instantiates a bus attachment for each of these devices. The bridge also exposes an AllJoyn bus attachment for itself with three bus objects: one which implements the AllJoyn standard About interface and two configuration interfaces, one for the Bridge and one for the Adapter.

Figure 6 shows the DSB overview with two pre-existing networks: Z-Wave and BACnet.

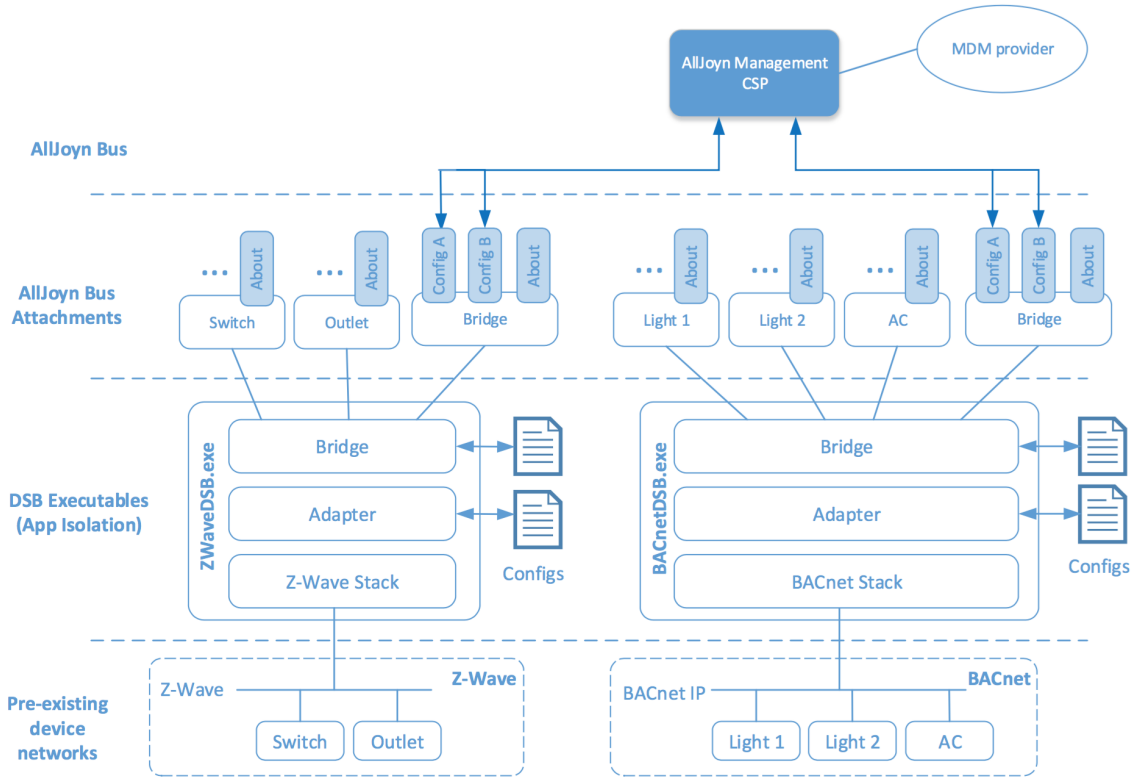


Figure 6: Device System Bridge Overview

The configuration interfaces provide settings to the DSB including:

- Access information for connecting to the alternate network and its devices.
- List of devices to be exposed as AllJoyn bus object.
- Security settings for each bus object.

3.1.2 *Configuration*

There are two separate configuration interfaces exposed by a DSB one for the Bridge and one for the Adapter. The configuration structure for the Bridge interface is the same for each DSB containing the bridge and device configuration. The structure of the Adapter is specific for each DSB containing e.g. access information for the pre-existing device network.

Bridge configuration

The XML file below shows an example of a Bridge configuration for three devices, only one of which is exposed to the AllJoyn bus. When a DSB runs the first time, a default configuration file is generated.

```
<?xml version="1.0" encoding="utf-8"?>
<BridgeConfig>
  <Settings>
    <Bridge>
      <KEYX></KEYX>
    </Bridge>
    <Device>
      <DefaultVisibility>false</DefaultVisibility>
      <KEYX></KEYX>
      <USERNAME></USERNAME>
      <PASSWORD></PASSWORD>
      <ECDHEECDSAPRIVATEKEY></ECDHEECDSAPRIVATEKEY>
      <ECDHEECDSACERTCHAIN></ECDHEECDSACERTCHAIN>
    </Device>
  </Settings>
  <Objects>
    <Object Id="001-001-001" Visible="false">
      <Desc>2 X Switch</Desc>
    </Object>
    <Object Id="001-002-001" Visible="true">
      <Desc>Dim Control 725</Desc>
    </Object>
    <Object Id="001-003-001" Visible="false">
      <Desc>Temperature Sensor 155</Desc>
    </Object>
  </Objects>
</BridgeConfig>
```

It includes authentication keys for Bridge and Device, and unique ID, visibility and description for each virtual device as it appears on the AllJoyn Network.

Adapter configuration

The sample XML below shows the configuration for a BACnet DSB adapter. The configuration details are Adapter specific and are defined by each adapter implementer.

```
<?xml version="1.0" encoding="UTF-8"?> <BACnetConfig>
<BACnetStack BBMD_IPAddress="xxx.xxx.xxx.xxx" BBMD_Port="yyyyy"
    NetworkInterface="" RequestPriority="8" DeviceInstanceMin="-1"
    DeviceInstanceMax="-1"/>
  <AllowedDeviceList>
    <Allowed>device_model_filter_token</Allowed>
  </AllowedDeviceList>
</BACnetConfig>
```

3.1.3 *Development*

Microsoft provides the *Visual Studio DSB Template*, an extension to Visual Studio that lets developers create new DSB projects [11]. It includes the DSB components such as the Bridge and a shell project for the adapter. These templates are provided to enable building AllJoyn Device System Bridges for Visual C# and Visual C++ projects.

The diagram in Figure 7 shows the classes developers will use in the Microsoft DSB template to create an abstraction of the native devices that need to be bridged into AllJoyn. The Bridge will use the instance of the adapter class to create the bus attachments for each device in the *Adapter.devices* list.

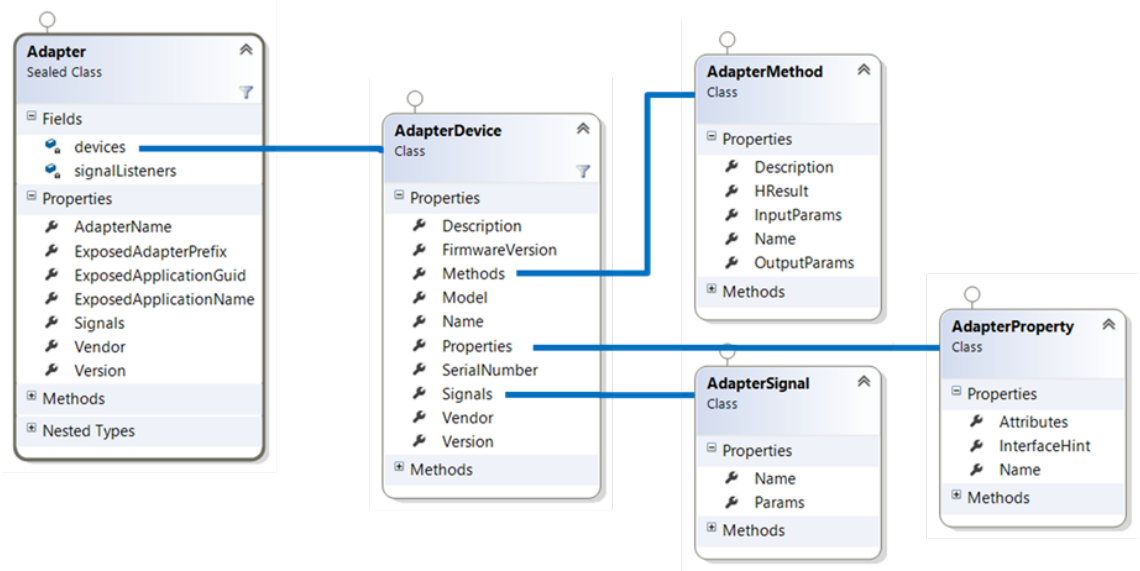


Figure 7: Class Diagram: DSB Adapter

Microsoft provides an API in order to map Bridge interface object to AllJoyn [12]. The main components are the following interfaces and methods.

IAdapter

An *IAdapter* represents the controller for a system of one or more devices that map to the AllJoyn bus. IAdapter declares interfaces necessary to support device enumeration, general configuration and life-cycle management. It also declares device properties, methods and signals. To expose a device as an AllJoyn service, it is necessary to implement a concrete class that inherits from IAdapter.

The IAdapter interface declares certain properties that must be implemented, like the application name and the application ID, that will be mapped to AllJoyn. The class also includes the *GetConfig* and *SetConfig* methods, used for accessing the adapter's configuration data described in section 3.1.2.

IAdapterDevice

From the bridge's perspective a device represents a device that the adapter implementer wants exposed to the AllJoyn bus as an AllJoyn Service. Each device has an AllJoyn interface for exposing all properties, method and signals encapsulated by the device.

IAdapterProperty, IAdapterMethod, IAdapterSignal

Set of the properties, the methods and the signals that a device exposes to AllJoyn. Properties and parameters are *IAdapterValue* classes.

3.1.4 *Limitations*

The main limitation of the DSB is the Microsoft environment, that constrains the bridge to run on Microsoft products. We preferred to develop our bridge in Java, in order to obtain a more portable and versatile product. Moreover, Java allows us to communicate with the CoAP network using Californium, a stable and powerful CoAP framework.

Microsoft developed its DSBs for some technologies like ZigBee [13] and BACnet [14]. We analyzed them and we noticed that each of them requires a pre-configuration that makes the bridge less versatile, especially on on-line device arrivals.

For example, in the Mock BACnet Adapter all devices need to be previously configured: device ID, name, vendor, description and the other device properties must be set in a device structure before the bridge starts to run. It also implies the possibility of inconsistency of data between the device descriptor and the physical device.

Otherwise, the ZigBee Adapter allows device arrivals and removals, but it maps devices using a dictionary. ZigBee adapter only supports some devices defined in ZigBee Light Link or Home Automation profiles. This means that it only implements the necessary ZCL clusters and ZDO commands to handle them. If an arriving device does not belong to a pre-defined cluster, it cannot be recognized by the bridge.

We prefer a more elastic solution, where every device that is supposed to be available for the AllJoyn network can be so, regardless of the arrival time and the kind of device this is. In the CoAP a device registers its resources to the bridge, and then they will be mapped into AllJoyn objects without the need of pre-configuration or device-specific dictionary.

3.2 COAP BRIDGE AS A DSB

Even if we preferred to develop the bridge in a more portable Java environment, it was possible to follow the Microsoft way and to develop a CoAP bridge as a DSB, for Microsoft products. AllJoyn objects and interfaces, and the methods and properties they implement, used to represent CoAP devices and resources, will be analyzed in a later

chapter. In the meantime, we can briefly show how to associate our bridge with the Device System Bridge, i.e. how to make a CoAP DSB using the Microsoft API.

We present two solutions that could be implemented. The first one uses the bridge device as an *IAdapterDevice* which contains all the CoAP resources; the second solution maps the CoAP devices as *IAdapterDevice* and each of them contains its own resources.

3.2.1 *Bridge device as IAdapterDevice*

The CoAP DSB could be developed with the bridge device mapped into an *IAdapterDevice*. In this way the devices in the AllJoyn network do not know anything about the CoAP devices and all the CoAP resources are presented by the bridge. However, this solution also has a lot of advantages: less information to be stored in the bridge, easier management of resources and freedom in on-line device arrivals and removals. The first statement is due to the fact that in this model the bridge has not to store neither information and configuration for each device it presents nor BusAttachment and other elements AllJoyn requires. The second and third statements are due to the fact that in the standard CoAP solution, the CoAP devices register to the Resource Directory giving only information about the resources they want to register, and here the focus is exactly on the resources and not on the devices.

In particular, a device which wants to provide its resources to the AllJoyn network, registers the resources in the Resource Directory and they are automatically and easily provided to the AllJoyn devices, without a previous configuration. The same thing happens during resource removals.

So, we opt to use a model like this one. In the Microsoft environment the bridge could be developed as follow:

- The *CoAPBridgeAdapter* is the main class of the CoAP adapter. This class derives from *IAdapter* and contains a *BridgeDevice* instance. *CoAPBridgeAdapter* class uses *CoAPAdapterSignal* to signal device arrival or removal.
- The only one device we see is the *BridgeDevice*. This class derives from the *IAdapterDevice* and contains a collection of *CoAPResource* instances.
- The *CoAPResource* class represents a CoAP resource provided by a CoAP device and it derives from *IAdapterProperty*. The class contains a collection of *CoAPMethod* instances, a collection of *CoAPProperty* instances and an istance of *CoAPSignal*. To be more accurate it contains a collection of method classes (*Get*, *Post*, *Delete*, *Registration* and *Cancellation* methods) which derive from the *CoAPMethod* abstract class. Similarly, it contains a collection of property classes (*InterfaceDescription* and *ResourceType* properties) which derive from the *CoAPProperty* abstract class. The signal class *Notification* represents the CoAP notification and it derives from the *CoAPSignal* abstract class.

- The *CoAPMethod* class is an abstract class that represents the CoAP methods. It derives from *IAdapterMethod*. This class contains a list of input and output parameters, that represent the request and response messages. These parameters are *IAdapterValue* classes.
- The *CoAPSignal* class is an abstract class that represents the CoAP notifications. It derives from *IAdapterSignal*. The notification message derives from *IAdapterValue*.
- The *CoAPProperty* class is an abstract class that represents the properties of a CoAP resource. It derives from *IAdapterAttribute*. This class contains an instance of *IAdapterValue* which contains the property value.
- The *CoAPAdapterSignal* class is used to handle notifications such as device arrival or removal. This class derives from *IAdapterSignal*.

3.2.2 *CoAP device as IAdapterDevice*

The second way to develop a CoAP DSB could be achieved by mapping each CoAP device as an *IAdapterDevice*. Using this solution the bridge presents to the AllJoyn network a virtual device for each CoAP device, and each of them contains its set of resources. This DSB is more similar to the DSBs that Microsoft developed for other technologies with respect to the first solution we presented, but it is not CoAP friendly.

The meaning of the last statement is for the fact that we cannot use the CoAP Resource Directory as it was defined, because now the bridge requires more information for each device that wants to register. Furthermore, the bridge has to contain a dictionary that allows to map all the device information, e.g. device manufactor, device model and so on; there is also the possibility that a device cannot be recognized and mapped, since it is not present in the dictionary. Microsoft used this DSB model in the ZigBee Adapter, where it only supports some devices defined in cluster profiles.

Otherwise, the bridge could present to the AllJoyn network only pre-configured devices. In this way no changes to the CoAP Resource Directory and on the CoAP devices are needed, but the bridge no longer supports on-line device arrivals and removals. Microsoft used this DSB model in the Mock BACnet Adapter, where each device must be configured before the bridge starts.

Due to the limitations described above, we preferred not to follow this model of bridge. However, it could be developed as follow:

- The *CoAPAdapter* is the main class of the CoAP adapter. This class derives from *IAdapter* and contains a collection of *CoAPDevice* instances. The *CoAPAdapter* class uses *CoAPAdapterSignal* to signal device arrival or removal.
- The *CoAPDevice* class represents a CoAP device. This class derives from the *IAdapterDevice* and contains a collection of *CoAPResource* instances.

- The *CoAPResource* class represents a CoAP resource provided by a CoAP device and it derives from *IAdapterProperty*. The class contains a collection of *CoAPMethod* instances, a collection of *CoAPProperty* instances and an istance of *CoAPSignal*. To be more accurate it contains a collection of method classes (*Get*, *Post*, *Delete*, *Registration* and *Cancellation* methods) which derive from the *CoAPMethod* abstract class. Similarly, it contains a collection of property classes (*InterfaceDescription* and *ResourceType* properties) which derive from the *CoAPProperty* abstract class. The signal class *Notification* represents the CoAP notification and it derives from the *CoAPSignal* abstract class.

- The *CoAPMethod* class is an abstract class that represents the CoAP methods. It derives from *IAdapterMethod*. This class contains a list of input and output parameters, that represent the request and response messages. These parameters are *IAdapterValue* classes.

- The *CoAPSignal* class is an abstract class that represents the CoAP notifications. It derives from *IAdapterSignal*. The notification message derives from *IAdapterValue*.

- The *CoAPProperty* class is an abstract class that represents the properties of a CoAP resource. It derives from *IAdapterAttribute*. This class contains an instance of *IAdapterValue* which contains the property value.

- The *CoAPAdapterSignal* class is used to handle notifications such as device arrival or removal. This class derives from *IAdapterSignal*.

3.2.3 *Configuration*

As it happens in the DSBs developed by Microsoft, the Adapter configuration file will contain the configuration details of the bridge with respect to the pre-existing device network, i.e. the CoAP network. It necessarily will include the device IP address and port.

4

DATA MAPPING

The key part of the bridge design is the data mapping process. It means that we have to find the best way to model a CoAP resource into an AllJoyn object, to choose how to represent CoAP messages into the AllJoyn network, and how the communication works between the two networks.

The Constrained Application Protocol (CoAP) is a web transfer protocol based on a REST architecture. CoAP provides a request/response interaction model between application endpoints and includes key concepts of the Web such as URIs and Internet media types. It is designed to easily interface with HTTP.

On the other hand, AllJoyn is an object-oriented software framework, based on the D-Bus specification. Objects have methods, signals and properties.

Having said that, it can be seen how out of touch the two technologies are and how many ways to map the data exist.

This chapter describes the data mapping process. It starts from the Resource Directory, that allows CoAP devices to register their resources. The bridge has to store the information that it needs in order to identify a device and to characterise a resource.

Then, we show how the request and response messages are represented, which means that CoAP messages must be translated into Java objects. It also includes a selection of CoRE attributes that should be provided to the AllJoyn applications, with the goal of an user-friendly experience.

The last section focuses on the AllJoyn objects that represent CoAP resources. In particular, the objects in the AllJoyn side have to implement the more suitable methods for a good modelling of a REST interface. The section also covers the implementation of the CoAP observing service in the AllJoyn network, using the signals the framework provides.

4.1 RESOURCE DIRECTORY

The discovery of resources offered by a constrained server is very important in machine-to-machine applications where there are no humans in the loop and static interfaces result in fragility.

Discovery of resources hosted by constrained web servers is specified by the CoRE Link Format [15]. This specification however only describes how to discover resources from the web server that hosts them by requesting *"/.well-known/core"*. In many M2M scenarios, direct discovery of resources is not practical due to sleeping nodes, disperse

networks, or networks where multicast traffic is inefficient. These problems can be solved by employing an entity called a *Resource Directory*, which hosts descriptions of resources held on other servers, allowing lookups to be performed for those resources.

A Resource Directory (*RD*) is used as a repository for resources hosted on other web servers, which are called endpoints (*EP*). An endpoint is a web server associated with a scheme, IP address and port (called *Context*), thus a physical node may host one or more endpoints. The RD implements a set of REST interfaces for endpoints to register and maintain sets of resource directory entries.

Endpoints are assumed to proactively register and maintain resource directory entries on the RD, which are soft state and need to be periodically refreshed. An endpoint is provided with interfaces to register, update and remove a resource directory entry. Furthermore, a mechanism to discover an RD using the CoRE Link Format is defined.

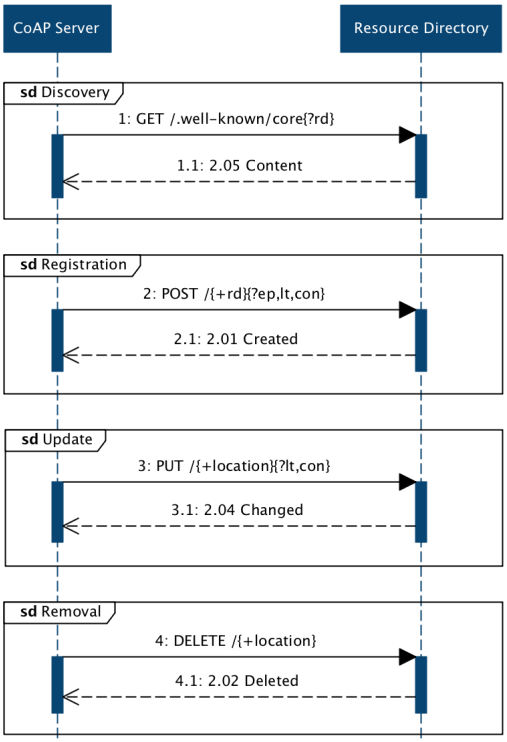


Figure 8: Sequence Diagram: Resource Directory

Figure 8 shows the sequence diagram for the Resource Directory. In particular, the diagram is divided into four phases: *Discovery*, *Registration*, *Update* and *Removal*.

4.1.1 *Discovery*

The REST interfaces between an RD and endpoints is called the *Resource Directory Function Set*. Before an endpoint can make use of an RD, it must first know the RD's IP address, port and the path of its RD Function Set.

Discovery is performed by sending either a multicast or unicast GET request to *"/.well-known/core"* and including a Resource Type (*rt*) parameter with the value *"core.rd"* in the query string. Upon success, the response will contain a payload with a link format entry for each RD.

The discovery request interface is specified as follows:

INTERACTION: $EP \rightarrow RD$

METHOD: *GET*

URI TEMPLATE: */.well-known/core{?rt}*

URI TEMPLATE VARIABLES:

- *rt*: Resource Type (optional). May contain the value *"core.rd"* or *"core.rd*"*.

CONTENT-TYPE: *application/link-format* (if any)

The following response codes are defined for this interface:

SUCCESS: *2.05 "Content"* with an *application/link-format* payload containing one or more matching entries for the RD resource.

FAILURE: *4.04 "Not Found"* is returned in case no matching entry is found for a unicast request.

FAILURE: *4.00 "Bad Request"* is returned in case of a malformed request for a unicast request.

FAILURE: No error response to a multicast request.

The following example shows an endpoint discovering an RD using this interface, thus learning that the base RD resource is, in this example, at */rd*.

```
REQ: GET coap://[ff02::1]/.well-known/core?rt=core.rd*

RES: 2.05 Content
     </rd>;rt="core.rd"
```

It is however expected that RDs will also be discoverable via other methods depending on the deployment, including assuming a default location (e.g. on an Edge Router in a LoWPAN), by assigning an anycast address to the RD or using DHCP.

4.1.2 *Registration*

After discovering the location of an RD Function Set, an endpoint may register its resources using the registration interface. This interface accepts a POST from an endpoint containing the list of resources to be added to the directory as the message payload. All parameters except the endpoint name are optional. The RD then creates a new resource or updates an existing resource in the RD and returns its location. An endpoint must use that location when refreshing registrations using this interface. Endpoint resources in the RD are kept active for the period indicated by the lifetime parameter. The endpoint is responsible for refreshing the entry within this period using either the registration or update interface. The registration interface must be implemented to be idempotent, so that registering twice with the same endpoint parameter does not create multiple RD entries.

The registration request interface is specified as follows:

INTERACTION: $EP \rightarrow RD$

METHOD: *POST*

URI TEMPLATE: */{+rd}{?ep,lt,con}*

URI TEMPLATE VARIABLES:

- *rd*: RD Function Set path (mandatory). This is the path of the RD Function Set, as obtained from discovery.
- *ep*: Endpoint name (mandatory). The endpoint name is an identifier that must be unique within a domain.
- *lt*: Lifetime (optional). Lifetime of the registration in seconds. Range of 60-4294967295. If no lifetime is included, a default value of 86400 (24 hours) should be assumed.
- *con*: Context (optional). This parameter sets the scheme, address and port at which this server is available in the form *scheme://host:port*. In the absence of this parameter the scheme of the protocol, source IP address and source port of the register request are assumed.

CONTENT-TYPE: *application/link-format*

The following response codes are defined for this interface:

SUCCESS: *2.01 "Created"*. This Location must be a stable identifier generated by the RD as it is used for all subsequent operations on this registration. The resource returned in the Location is only for the purpose of the Update (POST) and Removal (DELETE), and must not implement GET or PUT methods.

FAILURE: *4.00 "Bad Request"*. Malformed request.

FAILURE: *5.03 "Service Unavailable"*. Service could not perform the operation.

The following example shows an endpoint with the name *"node1"* registering two resources to an RD using this interface.

```
REQ: POST coap://rd.example.com/rd?ep=node1
     Content-Format:  40
     Payload:
     </sensors/temp>;ct=41;rt="temperature-c";if="sensor",
     </sensors/light>;ct=41;rt="light-lux";if="sensor"

RES: 2.01 Created
     Location:  /rd/4521
```

4.1.3 *Update*

The update interface is used by an endpoint to refresh or update its registration with an RD. To use the interface, the endpoint sends a POST request to the resource returned in the Location option in the response to the first registration. An update may update the lifetime or context parameters if they have changed since the last registration or update. Parameters that have not changed should not be included in an update. Upon receiving an update request, the RD resets the timeout for that endpoint and updates the scheme.

The update request interface is specified as follows:

INTERACTION: $EP \rightarrow RD$

METHOD: *POST*

URI TEMPLATE: /{+*location*}{?*lt*,*con*}

URI TEMPLATE VARIABLES:

- *location*: This is the Location path returned by the RD as a result of a successful earlier registration.
- *lt*: Lifetime (optional). Lifetime of the registration in seconds. Range of 60-4294967295. If no lifetime is included, a default value of 86400 (24 hours) should be assumed.
- *con*: Context (optional). This parameter sets the scheme, address and port at which this server is available in the form *scheme://host:port*. In the absence of this parameter the scheme of the protocol, source IP address and source port of the register request are assumed.

CONTENT-TYPE: *application/link-format*

The following response codes are defined for this interface:

SUCCESS: *2.04 "Changed"*. The update was successfully processed.

FAILURE: *4.00 "Bad Request"*. Malformed request.

FAILURE: *4.04 "Not Found"*. Registration does not exist (e.g. may have expired).

FAILURE: *5.03 "Service Unavailable"*. Service could not perform the operation.

The following example shows an endpoint updating its registration at an RD using this interface.

REQ: `POST /rd/4521`

RES: `2.04 Changed`

4.1.4 *Removal*

Although RD entries have soft state and will eventually timeout after their lifetime, an endpoint should explicitly remove its entry from the RD if it knows it will no longer be available (for example on shut-down). This is accomplished using a removal interface on the RD by performing a DELETE on the endpoint resource.

The removal request interface is specified as follows:

INTERACTION: $EP \rightarrow RD$

METHOD: *DELETE*

URI TEMPLATE: /{+*location*}

URI TEMPLATE VARIABLES:

- *location*: This is the Location path returned by the RD as a result of a successful earlier registration.

The following response codes are defined for this interface:

SUCCESS: *2.02 "Deleted"* upon successful deletion.

FAILURE: *4.00 "Bad Request"*. Malformed request.

FAILURE: *4.04 "Not Found"*. Registration does not exist (e.g. may have expired).

FAILURE: *5.03 "Service Unavailable"*. Service could not perform the operation.

The following examples shows successful removal of the endpoint from the RD.

REQ: DELETE /rd/4521

RES: 2.02 Deleted

4.2 RESOURCE DIRECTORY ENTRIES

4.2.1 *Entries format*

The resources obtained during the registration phase, as seen in 4.1.2, are kept in a database managed by the Resource Directory. Each resource involves inserting a new entry in the database. The entries remain in the database until the end of the lifetime specified at registration or when explicitly called the resource removal.

Each entry in the RD includes:

- *Endpoint Name*, that is the local unique identifier of the CoAP node;
- the *Endpoint Context*, i.e. address and port at which this server is available;
- the *Resource Path*, that identifies the resource within the device;
- the *Context Format*, in order to know the resource representation format;
- the *Resource Type* and *Interface Description* strings that act as information for clients;
- the resource *Lifetime*, at the end of which the resurce is no longer valid;
- the *Location*, that uniquely represents the device within the RD.

Table 6 shows the database entries format.

NAME	QUERY	TYPE	DESCRIPTION
Endpoint Name	ep	string	Unique local name
Endpoint Context	con	ip:port	EP IP address and port
Location	location	string	Used by the EP for Update and Removal
Resource Path	rd	string	Path of the resource in the EP
Context Format	ct	int	Resource representation
Resource Type	rt	string	Opaque string for resource description
Interface Description	if	string	Opaque string for interface description
Lifetime	lt	int	Lifetime of the resource in seconds

Table 6: RD Entry Format

Not all the fields are mandatory. In particular, *ep, con, rd* and *location* are mandatory, the other fields are optional. If optional fields are not present, default values are assigned: *rt* and *if* have a null string as default value, and *lt* is set to 24 hours.

A resource is uniquely identified within the Resource Directory by the pair *location* and *rd* since they uniquely identify the device and the resource within the device. So, this pair serves as the primary key of the resource.

Each entry in the database has a timer initialized at *Lifetime*, at the end of which the resource will be removed.

4.2.2 *Storage of resources*

Currently, the entry fields are stored in several hash maps in a volatile manner, but they can be easily stored in a non-volatile way by the implementation of a database. In particular, there are five hash maps:

- A map containing the *(identifier, context)* pair for each registered node. The *identifier* is the local unique identifier of the CoAP node specified during the registration phase.
- A map containing the *(resource, node)* pair for each registered resource, where *resource* is the resource path within the RD.
- A map containing the *(resource, type)* pair for each registered resource. The *type* is the resource type field *rt* set during the resource registration, if specified.
- A map containing the *(resource, interface)* pair for each registered resource. The *interface* is the interface description field *if* set during the resource registration, if specified.
- A map containing the *(resource, path)* pair for each registered resource. The two fields are, respectively, the resource path within the RD and the resource path within the node.

4.3 COAP MESSAGES IN THE ALLJOYN NETWORK

CoAP request and response semantics are carried in CoAP messages. The messages that are exchanged in the AllJoyn network should carry request and response messages of the same format specified in the Constrained Application Protocol [2].

4.3.1 *Message fields*

Not all the CoAP message fields are useful to the AllJoyn application, thus we analyze which fields should be included and which not:

VERSION: It must be set to *1*, otherwise the message will be ignored. Not included.

TYPE: Indicates if this message is of type *Confirmable*, *Non-confirmable*, *Acknowledgement*, or *Reset*. It is used in the CoAP side and it regards a lower level respect to AllJoyn. Not included.

TOKEN LENGTH: Indicates the length of the variable-length *Token* field, not used here. Not included.

CODE: In case of a request, the *Code* field indicates the Request Method; in case of a response, a Response Code. Included.

MESSAGE ID: Used to detect message duplication. As for the *Type* field, it regards a lower level. Not included.

TOKEN: The Token value is used to correlate requests and responses. It regards a lower level, again. Not included.

OPTIONS: Message options, explained later. Included.

PAYLOAD: The body data. Included.

The included fields and the types with which they are represented are shown in Table 7. The correspondence between AllJoyn type ID (used in Table 7) and AllJoyn type is shown in Table 13.

FIELD	AJ TYPE	JAVA TYPE
Code	u	enum
Options	r	Options
Payload	ay	byte[]

Table 7: Message Fields

4.3.2 *Option fields*

Both requests and responses may include a list of one or more options. The options in the *Options* field and their usefulness in AllJoyn are the following:

CONTENT-FORMAT: It indicates the representation format of the message payload. The representation format is given as a numeric *Content-Format* identifier. Included.

ETAG: It is a resource-local identifier for differentiating between representations of the same resource that vary over time. The *ETag* option in a response provides the current value of the entity-tag for the "tagged representation". In a GET request, an endpoint that has one or more representations previously obtained from the resource, and has obtained *ETag* response options with these, can specify an instance of the *ETag* option for one or more of these stored responses. Included.

LOCATION-PATH & LOCATION-QUERY: The two options together indicate a relative URI that consists either of an absolute path, a query string, or both. A combination of these options is included in a *2.01 (Created)* response to indicate the location of the created resource. Each *Location-Path* option specifies one segment of the absolute path to the resource, and each *Location-Query* option specifies one argument parameterizing the resource. Not included.

MAX-AGE: It indicates the maximum time a response may be cached before it is considered not fresh. Caching is managed by the Bridge. Not included.

PROXY-URI & PROXY-SCHEME: They are used to make a request to a forward-proxy. Proxying is managed by the Bridge. Not included.

URI-HOST, URI-PORT, URI-PATH, & URI-QUERY: They are used to specify the target resource of a request to a CoAP origin server. Since host, port and path are managed by the Bridge, they are not used in AJ messages. About the *Uri-Query* field, the query filtering will be handled in a more user-friendly manner. Not included.

ACCEPT: It can be used to indicate which *Content-Format* is acceptable to the client. The server returns the preferred *Content-Format* if available. If the preferred *Content-Format* cannot be returned, then a *4.06 "Not Acceptable"* will be sent as a response. Included.

IF-MATCH & IF-NONE-MATCH: The first one may be used to make a request conditional on the current existence or the value of an *ETag* for one or more representations of the target resource. The second one may be used to make a request conditional on the nonexistence of the target resource. Included.

SIZE1: It provides size information about the resource representation in a request. Included.

OBSERVE: When included in a GET request, it extends the GET method so it does not only retrieve a current representation of the target resource, but also requests the server to add or remove an entry in the list of observers of the resource. When included in a response, the *Observe* option identifies the message as a notification. Here the observing service will be implemented in a more user-friendly manner. Not included.

Table 8 lists the included options and their types.

OPTION	AJ TYPE	JAVA TYPE
Content-Format	u	int
ETag	ay	byte[]
Accept	u	int
If-Match	ay	byte[]
If-None-Match	b	boolean
Size1	u	int

Table 8: Option Fields

Since it may be present more than one ETag - and then more than one If-Match field - and due to the problems that may arise in AllJoyn during data marshalling/unmarshalling with arrays of arrays of bytes, we preferred to store the ETags and the If-Match fields as strings.

We represent the options as a class:

```
public class Options {

   public int contentFormat;
   public String[] etags;
   public int accept;
   public String[] ifMatch;
   public Boolean ifNoneMatch;
   public int size1;

}
```

Since AllJoyn cannot recognize the parameters signature during data marshalling if they contain private fields, all the classes we use must have public attributes. It affects the options class as well as the request and the response messages.

4.3.3 *Query filtering*

The syntax of a CoAP URI Scheme is the follow one:

coap-URI = "coap:" "//" host [":" port] path ["?" query]

It is composed by the *Uri-Host*, the *Uri-Port*, the *Uri-Path* and the *Uri-Query*, where each option holds the following values:

- The *Uri-Host* option specifies the Internet host of the resource being requested. It is managed by the AllJoyn framework and the Bridge.

- The *Uri-Port* option specifies the transport-layer port number of the resource. It is managed by the AllJoyn framework and the Bridge.
- The *Uri-Path* option specifies a segment of the absolute path to the resource. It is managed by the Bridge.
- Each *Uri-Query* option specifies one argument parameterizing the resource.

The *Uri-Query* field in the option list is used to perform query filtering. The query serves to further parameterize the resource. It consists in a sequence of arguments separated by the *"&"* character. An argument is in the form of a *"key=value"* pair. Since in CoAP the query is represented by a string, it would be better to do query filtering in a more user-friendly way in AllJoyn using a dictionary of *key-value* pairs (Table 9).

AJ TYPE	JAVA TYPE
a{ss}	Map<String,String >

Table 9: Query Attributes

Table 10 shows an example of transition from the URI to the query attributes.

URI	KEY	VALUE
/.well-known/core?rt=light-lux	rt	light-lux
/.well-known/core?rt=light-lux&if=sensor&ct=41	rt if ct	light-lux sensor 41

Table 10: Example: Query Attributes

An AllJoyn application specifies zero or more query options, the Bridge composes the URI from them, and then it sends the message to the CoAP Server.

4.3.4 *Request message*

A request is initiated by setting the *Code* field in the CoAP header to a Method Code and including request information. The *Code* field is a 8-bit unsigned integer, split into a 3-bit class (most significant bits) and a 5-bit detail (least significant bits). The class can indicate a request (*0*), a success response (*2*), a client error response (*4*), or a server error response (*5*).

In the Request Message the *Code* field can assume one of the *CoAP Method Codes* in Table 11.

CODE	NAME	INTEGER VALUE
0.01	GET	1
0.02	POST	2
0.03	PUT	3
0.04	DELETE	4

Table 11: Method Codes

The request message does not need the *Code* field in AllJoyn, since the code is assigned by the Bridge according to the called method (e.g., the *Get* method corresponds to the GET code). Therefore, the Request Message includes the message fields (4.3.1), the options (4.3.2) and the query attributes (4.3.3).

The *CoAPRequestMessage* Java interface is used in the request messages.

```
public interface CoAPRequestMessage {

   public Options getOptions();
   public void setOptions(Options options);

   public Map<String,String> getAttributes();
   public void setAttributes(Map<String,String> attributes);

   public byte[] getPayload();
   public String getPayloadString();
   public void setPayload(byte[] payload);
   public void setPayload(String payload);

}
```

The *CoAPRequestMessage* interface is implemented by the *RequestMessage* class.

4.3.5 *Response message*

After receiving and interpreting a request, a server responds with a CoAP response. A response is identified by the *Code* field in the CoAP header being set to a Response Code. As descibed in 4.3.4, the upper three bits of the 8-bit *Code* field define the class of response. There are three classes of Response Codes:

2 - SUCCESS: The request was successfully received, understood, and accepted.

4 - CLIENT ERROR: The request contains bad syntax or cannot be fulfilled.

5 - SERVER ERROR: The server failed to fulfill an apparently valid request.

We simply consider the code as an unsigned integer, without its division into class and detail. The *Code* field in the *ResponseMessage* class can assume one of the *CoAP Response Codes* in Table 12.

CODE	NAME	INTEGER VALUE
2.01	Created	65
2.02	Deleted	66
2.03	Valid	67
2.04	Changed	68
2.05	Content	69
4.00	Bad Request	128
4.01	Unauthorized	129
4.02	Bad Option	130
4.03	Forbidden	131
4.04	Not Found	132
4.05	Method Not Allowed	133
4.06	Not Acceptable	134
4.12	Precondition Failed	140
4.13	Request Entity Too Large	141
4.15	Unsupported Content-Format	143
5.00	Internal Server Error	160
5.01	Not Implemented	161
5.02	Bad Gateway	162
5.03	Service Unavailable	163
5.04	Gateway Timeout	164
5.05	Proxying Not Supported	165

Table 12: Response Codes

The response code is represented by the *ResponseCode* enumerated type, contained in the *CoAP* class.

```
public enum ResponseCode {

   CREATED(65),
   DELETED(66),
   VALID(67),
   CHANGED(68),
   CONTENT(69),
   CONTINUE(95),
   BAD_REQUEST(128),
   UNAUTHORIZED(129),
   BAD_OPTION(130),
   FORBIDDEN(131),
   NOT_FOUND(132),
   METHOD_NOT_ALLOWED(133),
   NOT_ACCEPTABLE(134),
   PRECONDITION_FAILED(140),
```

```
   REQUEST_ENTITY_TOO_LARGE(141),
   UNSUPPORTED_CONTENT_FORMAT(143),
   INTERNAL_SERVER_ERROR(160),
   NOT_IMPLEMENTED(161),
   BAD_GATEWAY(162),
   SERVICE_UNAVAILABLE(163),
   GATEWAY_TIMEOUT(164),
   PROXY_NOT_SUPPORTED(165);

}
```

In addition to the code, the *ResponseMessage* includes the message fields (4.3.1) and the options (4.3.2).

The *ResponseMessage* class implements the *CoAPResponseMessage* Java interface.

```
public interface CoAPResponseMessage {

   public CoAP.ResponseCode getCode();
   public void setCode(CoAP.ResponseCode code);

   public Options getOptions();
   public void setOptions(Options options);

   public byte[] getPayload();
   public String getPayloadString();
   public void setPayload(byte[] payload);
   public void setPayload(String payload);

}
```

4.4 COAP RESOURCES IN THE ALLJOYN NETWORK

The resources obtained should be available to the AllJoyn network. To achieve this, CoAP resources need to be mapped in AJ objects, creating an association between the data in the database and the features that characterize objects like interfaces, path, methods, properties.

4.4.1 *Setting AllJoyn interface, methods and properties*

The AllJoyn framework enables inter-process communication through an object. The object is defined as a bus interface. An interface can contain methods, signals and properties.

```
@BusInterface (name = "org.my.interface.name")
public interface MyInterface {
```

```
    @BusMethod
    public String MyMethod(String inStr) throws BusException;

    @BusSignal
    public void MySignal(String inStr) throws BusException;

    @BusProperty
    public String GetMyProperty() throws BusException;

    @BusProperty
    public void SetMyProperty(String myProperty) throws BusException;

}
```

The *@BusInterface* annotation tells the code that this interface is an AllJoyn interface. All bus interfaces must have a name. If no name is specified, a default name is assigned. The default interface name is *".".*

Interface naming rules follow the D-Bus specification.

The *@BusMethod* annotation tells the Java compiler that this is a bus method. AllJoyn methods work almost equal to a regular method in Java. The major difference is that the AllJoyn methods execute on a different process or device. The method can accept multiple arguments and reply with multiple arguments.

The *@BusMethod* annotation has four properties: *annotation*, *name*, *signature*, and *replySignature*. Under normal circumstances, the values for the annotation properties can be determined by the AllJoyn framework. However, there are instances in which the signature must be specified: for example an unsigned integer must be sent in a method. Since Java does not have an unsigned integer type, this must be specified in the *@BusMethod* annotation.

```
@BusMethod(signature="u", replySignature="u")
public int MyMethod3(int unsignedArg) throws BusException;
```

The valid values for the *signature* and *replySignature* are the set of values that are valid according to the D-Bus specification.

The *@BusSignal* annotation specifies that the following code is an AllJoyn signal. Unlike methods, signals have no replies. For this reason signals always have a return type of void. Like methods, signals can take multiple arguments.

A signal is seen only if a program has registered a signal handler for that signal.

The *@BusProperty* annotation specifies that the following code is an AllJoyn property. AllJoyn properties are exactly like AllJoyn methods except they are specialized for get/set

commands of a single value.

Beyond simple data types, the AllJoyn framework can handle complex data types such as arrays, maps, and structs. In the case of arrays and maps, the data type can be handled by the AllJoyn code with no special action. However, structs require additional annotation. The AllJoyn framework must know the order of all elements of a struct so that it can marshal and unmarshal the message. This is where the *@Position* annotation is used.

```
public class MyStruct{
 @Position(0)
 public String name;
 @Position(1)
 public int valueOne;
 @Position(2)
 public int valueTwo;
}
```

The *@Position* annotation numbering must start from *"0"* (zero) and count up in increments of one. Skipping a number, like going from *@Position(1)* to *@Position(3)* without having an *@Position(2)* anywhere in the code, is a logic error.

Table 13 shows valid AllJoyn data types which *signature* and *replySignature* can assume, and compatible Java types. The signature for all structures is *"r"* or the list of all the values of the structure inside parentheses. For example, the signature for the *MyStruct* class would be *"(sii)"*. Structs can be nested.

Since all CoAP resources have the same REST methods and the same attributes, the same interface is implemented by all the objects, and then by all the resources. Furthermore, to compute discovery an application either should know in advance the interface name or must invoke introspection to know the methods and properties the interface is made of. Therefore, there is no need to know the resource by the interface name (e.g. temperature sensor or light sensor) because all of them are REST interfaces and the resources they are related to will be shown by the object path and the resource properties *rt* and *if*.

In the default scheme the interface implements the *get*, *post* and *delete* methods. The *signature* and the *reply signature* of the first one and of the second one are *structs*, according to the *RequestMessage* and the *ResponseMessage* classes described in 4.3, since the messages are composed by the payload, the options and the attributes. The same is true for the signature of the *registration* method and the *notification* signal, and for the reply signature of the *delete* method. We used the classes as parameters instead of the interfaces because AllJoyn could not recognize the fields from which the messages are composed.

In addition to them, the interface implements the *registration* and *cancellation* methods and the *notification* signal used in the observing service. These methods uses the client

TYPE ID	ALLJOYN TYPE	COMPATIBLE JAVA TYPES
y	byte	byte
b	boolean	boolean
n	int16	short
q	uint16	short
i	int32	int
u	uint32	int
x	int64	long
t	uint64	long
d	double	double
s	string	String
o	object_path	String
g	signature	String
a	array	Array
r	struct	User-defined type
v	variant	Variant
a{TS}	dictionary	Map< JT,JS>

Table 13: AllJoyn and Java compatible data types

application unique name as parameter in order to avoid duplicate registrations from the same client. The *registration* function returns an integer as a status value, in which it specifies the success of the observing registration.

Finally, the *CoAPInterface* implements two properties, *ResourceType* and *InterfaceDescription*, which allow to read the resource type and the interface description attributes of the resource.

```
@BusInterface (name="com.bridge.Coap", announced="true")
public interface CoAPInterface {

   @BusMethod (name="get", signature="r", replySignature="r")
   public ResponseMessage get(RequestMessage request) throws BusException;

   @BusMethod (name="post", signature="r", replySignature="r")
   public ResponseMessage post(RequestMessage request) throws BusException;

   @BusMethod (name="delete", replySignature="r")
   public ResponseMessage delete() throws BusException;

   @BusMethod (name="registration", signature="sr", replySignature="i")
   public Status registration(String uniqueName, RequestMessage request) throws
       BusException;

   @BusMethod (name="cancellation", signature="s")
```

```
    public void cancellation(String uniqueName) throws BusException;

    @BusSignal (name="notification", signature="r")
    public void notification(ResponseMessage message) throws BusException;

    @BusProperty (name="ResourceType")
    public String getResourceType() throws BusException;

    @BusProperty (name="InterfaceDescription")
    public String getInterfaceDescription() throws BusException;

}
```

The definition of the `com.bridge.Coap` interface is reported in Table 14.

NAME	TYPE	PARAMETERS	REPLY
`get`	method	`RequestMessage`	`ResponseMessage`
`post`	method	`RequestMessage`	`ResponseMessage`
`delete`	method	none	`ResponseMessage`
`ResourceType`	property	none	String
`InterfaceDescription`	property	none	String
`registration`	method	uniqueId, `RequestMessage`	Status
`cancellation`	method	uniqueId	none
`notification`	signal	`ResponseMessage`	none

Table 14: `com.bridge.Coap` interface

Setting the signature value to *struct* does not means that AllJoyn can recognize the field from which the parameters are composed. That value indicates an user-defined type, but you have to specify the signatures of the elements inside the classes. In this context, we set the following signature for the classes we used:

- The *Option* class is composed, as described in 4.3.2, by:
  - an integer,
  - an array of strings,
  - an integer,
  - an array of strings,
  - a boolean,
  - an integer.

  So, its signature will be *(iasiasbi)*.

- The *RequestMessage* class, described in 4.3.4, is composed by:
  - an Option class,
  - a map with (string, string) pairs,
  - an array of bytes.

  So, its signature will be *(ra{ss}ay)*, that AllJoyn will interpret as *((iasiasbi)a{ss}ay)*.
- The *ResponseMessage* class, described in 4.3.5, is composed by:
  - an integer,
  - an Option class,
  - an array of bytes.

  So, its signature will be *(iray)*, that AllJoyn will interpret as *(i(iasiasbi)ay)*.

4.4.2 *Setting AllJoyn object path*

When connecting to the bus, a program may act as a service, a client, or both.

Connecting a service consists on the creation of a new BusAttachment, on the registration of a BusObject with a given absolute path using the BusAttachment and on the connection of the BusAttachment to the bus.

```
mBus = new BusAttachment("applicationName");
mBus.registerBusObject(this, "/servicepath");
mBus.connect();
```

Resources are represented by instances of objects of the same type that implements the same interface defined in 4.4.1. Each instance is identified by the object path that has to be unique local and is composed by:

- *Location*;
- *Resource Path* (*rd*).

The two attributes guarantee uniqueness, since *location* is a stable CoAP Server identifier, unique within the Resource Directory and *rd* is unique within the CoAP Server.

Using the example in 4.1.2 the object paths obtained are shown in Table 15.

LOCATION	RESOURCE PATH	OBJECT PATH
/rd/4521	/sensors/temp	/rd/4521/sensors/temp
/rd/4521	/sensors/light	/rd/4521/sensors/light

Table 15: Example: Object Path

4.4.3 *Implementing the observing service*

A CoAP resource may be observable. It means that a client interested in that resource can register to the observing service and then the CoAP server will send him notifications containing the resource value.

The corresponding CoAP observing service is implemented in AllJoyn by *Signals*. In particular, the resource observing service is divided in two phases:

- registration phase;
- notification phase.

In the registration phase, an AllJoyn application interested into observing a resource calls the *Registration* method on the object representing that resource. In the CoAP side it corresponds to a resource registration via extended GET request. The bridge sends the GET request to the server with the observe field set to *o* (register), and waits for a response: if the response message has the observe field set, it means that the server implements the observing service, otherwise it don't. The registration method returns the status *OK* if the resource is observable, and the status *NOT_IMPLEMENTED* if the resource is not observable. Once almost one AJ device is interested into observe a resource, each time the resource sends new data, it is forwarded via signal (notification phase).

The *CoAPInterface* interface in the AllJoyn object implements the *Notification* signal. Signals, unlike methods, never return a value. Client applications that are interested in receiving a signal must register for that signal with the bus. The client should implement a signal handler to respond to the signal for which it was registered.

In the service side, a *SignalEmitter* is required to emit a signal. Once a *SignalEmitter* is created, an interface can be made to send the actual signals. No coding is needed to emit the signals beyond defining and using the interface.

Considering an object that implements the interface *"MyInterface"*, used in 4.4.1, the emission of a signal will be as follows:

```
SignalEmitter emitter = new SignalEmitter(myObject, joinerName, sessionId,
                        SignalEmitter.GlobalBroadcast.Off);
myInterface = emitter.getInterface(MyObject.class);

myInterface.MySignal("message");
```

In the client side, when registering for signal handlers, a class should contain a method with the *@BusSignalHandler* annotation.

```
@BusSignalHandler(iface="org.my.interface.name", signal="MySignal")
public void MySignal(String inStr) {
   // handle signal
```

```
}
```

The important part of the *BusSignalHandler* is to get the *iface* name and the signal arguments correct. If these are not correct, it does not catch the emitted signal.

The observing service could be done both with sessionful and sessionless signals.

The way to send sessionless signals is not much different than sending a regular signal. Only the setting of the the flag specifying that it is a sessionless signal and the setting of the session ID used to send the signal to *o* are needed. A sessionless signal does not need a session id. The two things to know in the case of a sessionless signal are:

- It has a session ID of *o*.
- It has the sessionless flag set to indicate that it is a sessionless signal.

We prefer to have signals associated with sessions. In this way the bridge can choose the destination to which it sends the notification, and only registered clients can receive it. In order to implement it, there is a *SignalEmitter* for each pair composed by application unique name and session ID.

On the consumer side, the application registers an interest in a signal by calling the D-Bus *AddMatch* method. This must be done during the registration phase.

```
Status status = mBus.addMatch("interface=org.my.interface.name'");
```

Rules are specified as a string of comma separated key/value pairs. Multiple key/-value pairs may be specified in a single match rule. These are treated as a logical AND when discovering signal providers. The exclusion of a key from the rule indicates a wildcard match. Table 16 describes some keys that can be used to create a match rule.

KEY	POSSIBLE VALUES	DESCRIPTION
type	*'signal'*, *'method_call'*, *'method_return'*, *'error'*	Match on the message type.
sender	A bus or unique name	Match messages sent by a particular sender.
interface	An interface name	Match messages sent over or to a particular interface.
path	An object path	Matches messages which are sent from or to the given object.

Table 16: Match Rule Keys

For instance, if an AJ client is interested into receiving notifications from a CoAP resource with object path */rd/4521/sensors/temp*, it adds the following math rule:

$$"interface =' com.bridge.Coap', path =' /rd/4521/sensors/temp'"$$

An AllJoyn application interested in receiving notification from an object, first calls the *registration* method on that object, then adds the match rule for the *"com.bridge.Coap"* interface and the interested object path.

The application unregisters interest in a resource observation by calling the *cancellation* method on the AllJoyn object. Then the application calls the *removeMatch* method with a previously added match rule to remove the reception of a signal.

When no applications are interested into receiving resource notification, the Bridge informs the CoAP Server that it wants to be removed from the list of observers for that resource. This is done either rejecting a notification with a RST message or performing a GET request that has the *Token* field set to the token of the observation to be cancelled and includes an *Observe Option* with the value set to *1* (deregister) [16].

4.4.4 *About data*

The *About* interface is to be implemented by an application on a target device. This interface allows the application to advertise itself so other applications can discover it. A client can discover the application via an announcement which is a sessionless signal containing the basic application information like application 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 application provides functionality of interest.

The bridge implements the About interface and it sends about data every time a CoAP resource registers or unregisters to it. The about data contains the *AppId* and the *AppName*, the software version, the description and the other required information. It contains also the list of registered *CoAPResource* objects.

An AllJoyn client interested into the available CoAP resource in the network, inspects the about data it receives and then it knows all the objects the bridge offers. Using the object paths contained in the about data, the AJ client can obtain the proxy objects representing the resources it wants, and then it can call methods on them.

4.4.5 *Introspection file*

The formal definition of an AllJoyn Interface is expressed in a well-defined XML format, called the Introspection XML language. The name is due to the fact that the definition format was originally designed for run-time introspection of D-Bus (and later AllJoyn) bus objects and their Interfaces.

The interfaces that are available to the AJ network follow the form described in the introspection file using the Introspection XML language. The interface definition specifies the interface name, the implemented methods and properties and their names.

```
<node>
  <interface name="com.bridge.Coap">
    <description>RESTful CoAP interface</description>
    <method name="get">
      <description>Send a GET method call</description>
      <arg name="request" type="((iasiasbi)a{ss}ay)" direction="in"/>
      <arg name="response" type="(i(iasiasbi)ay)" direction="out"/>
    </method>
    <method name="post">
      <description>Send a POST method call</description>
      <arg name="request" type="((iasiasbi)a{ss}ay)" direction="in"/>
      <arg name="response" type="(i(iasiasbi)ay)" direction="out"/>
    </method>
    <method name="delete">
      <description>Send a DELETE method call</description>
      <arg name="response" type="(i(iasiasbi)ay)" direction="out"/>
    </method>
    <method name="registration">
      <description>Start to observe the resource</description>
      <arg name="uniqueName" type="s" direction="in"/>
      <arg name="request" type="((iasiasbi)a{ss}ay)" direction="in"/>
      <arg name="status" type="i" direction="out"/>
    </method>
    <method name="cancellation">
      <description>Stop to observe the resource</description>
      <arg name="uniqueName" type="s" direction="in"/>
    </method>
    <signal name="notification" sessionless="false">
      <description>A notification arrived</description>
      <arg name="message" type="(i(iasiasbi)ay)" direction="out"/>
    </signal>
    <property type="s" access="read">
      <description>The Resource Type field</description>
    </property>
    <property type="s" access="read">
      <description>The Interface Description field</description>
    </property>
  </interface>
  <interface name="org.freedesktop.DBus.Introspectable">
    <method name="Introspect">
      <arg name="data" type="s" direction="out"/>
    </method>
  </interface>
  <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>
    <annotation name="org.alljoyn.Bus.Secure" value="off"/>
  </interface>
</node>
```

The interfaces described using the Introspection XML language is validated against an XML scheme.

Introspect is a method call. This method call is a member of *org.freedesktop.DBus.Introspectable* interface. It takes no input arguments and returns a string in XML format, as described in the D-Bus Specification.

Using the Introspection XML file, a client could use the AllJoyn Code Generator [17] in order to obtain code ready for compilation and running, without the need to know in advance the CoAP resource interface.

5

SYSTEM DESIGN

The *Unified Modeling Language* (UML) is a standard visual modelling language intended to be used for analysis, design, and implementation of software-based systems. UML specification defines two major kinds of UML diagram:

- Structure diagrams, which show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other.
- Behaviour diagrams which show the dynamic behaviour of the objects in a system, which can be described as a series of changes to the system over time.

The UML standard provides over than twenty diagram to design and model a software.

In this chapter we shows how the bridge was design using three structure diagrams, and how the components interact each other using a behaviour diagram. About the structure ones, the chapter includes the deployment diagram, the component diagram, and the class diagrams. About the behaviour, the chapter shows a sequence diagram for each use case we met.

5.1 DEPLOYMENT DIAGRAM

The deployment diagram in Figure 9 models the physical deployment of artifacts (software components) on nodes (hardware components). A device node is a physical computational resource with processing capability upon which artifacts may be deployed for execution. An artifact is the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system. Artifacts are the physical entities that are deployed on nodes.

Our scheme is composed by the following nodes:

- The *AllJoyn Device*, which belongs to the AllJoyn network. An AllJoyn network can be composed by one or more AllJoyn devices. The AJ Device can be a computer, a smartphone or a smart object able to run the AllJoyn framework (e.g., a smart TV, a smart refrigerator). The AJ Device contains two artifacts: the *AllJoyn Client Application* and the *AllJoyn framework*. On a single device should run more than one AJ application.
- The *Bridge Device*, which is located between the two networks. A single Bridge device is sufficient to connect the two networks and to map the CoAP resources

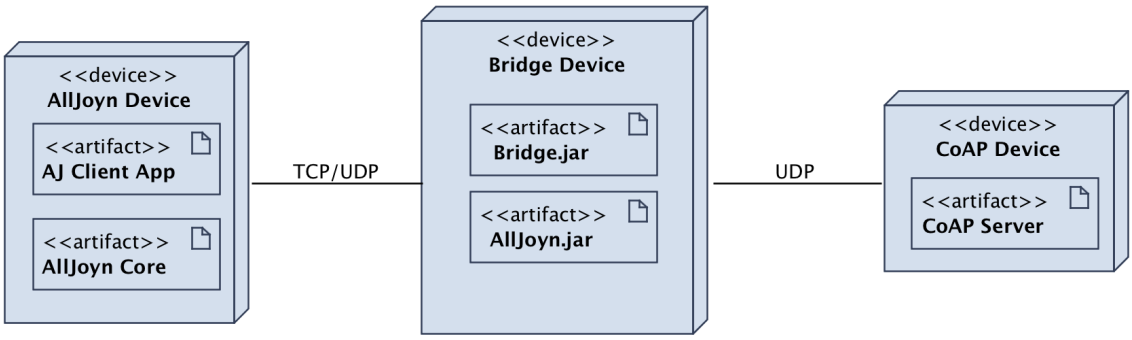


Figure 9: Deployment Diagram

into AJ objects. A Bridge device can be any device able to support the AllJoyn framework. It contains the *AJ framework* and the *Bridge application*.

- The *CoAP Device*, which belongs to the CoAP network. A CoAP network can be composed by one or more CoAP devices. Tipically, CoAP devices are constrained nodes, which often have 8-bit microcontrollers with small amounts of ROM and RAM. CoAP networks are constrained (e.g., low-power, lossy) networks, such as IPv6 over Low-Power Wireless Personal Area Networks (6LowPANs), which often have high packet error rates and throughput of 10s of Kbit/s. The CoAP Device contains the *CoAP Server application*.

The nodes are interconnected through communication paths by which they are able to exchange signals and messages. The communication between an AllJoyn device and the Bridge device can be done either via reliable connection (TCP) or unreliable connection (UDP). The communication between a CoAP device and the Bridge device is done via UDP connection.

5.2 COMPONENT DIAGRAM

The component diagram describes how the software system is split up into components and shows the dependencies among them. It is shown in Figure 10.

The main components are the *AJ Client* inside the AJ device, the *Bridge* and the *CoAP Server* inside the CoAP device. The AJ Client contains both the following:

- The *AllJoyn Application* that acts as the client application.
- The *AllJoyn Core Library*, which provides the lowest level set of APIs to interact with the AllJoyn network and with the Bridge.

The Bridge has five sub-components:

- The *Resource Directory*, which accepts requests from CoAP resources and stores their information.

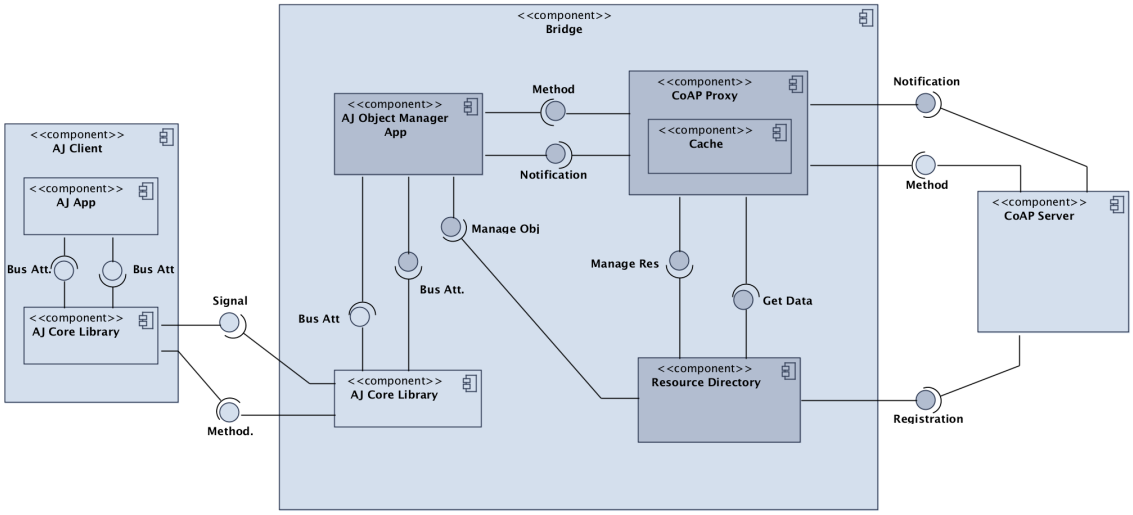


Figure 10: Component Diagram

- The *CoAP Proxy*, which interacts with the other components during the resource registration, resource removal, method invocation and notification, and provides caching.
- The *Cache*, inside the *CoAP Proxy*, which stores data to make future requests for that data served faster.
- The *AllJoyn Object Manager Application*, which interacts with the AllJoyn network and provides it the CoAP resources in the form of AJ objects.
- The *AllJoyn Core Library*, which provides the lowest level set of APIs to interact with the AllJoyn network.

5.3 CLASS DIAGRAMS

The class diagram is a type of static structure diagram that describes the structure of a system by showing the system classes, their attributes, methods, and the relationships among objects. The class diagrams for each of the component we have implemented are shown below.

5.3.1 *Bridge*

The Bridge is the main class of the project. It implements the *main* method in which it instantiates and starts the other components.

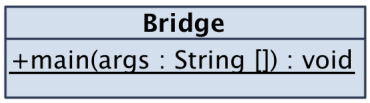


Figure 11: Class Diagram: Bridge

5.3.2 *Resource Directory*

The Resource Directory is composed by its main class, which contains the maps that are used to store registered resource information, and it implements the methods used to manage RD entries insertion and removal.

The *ResourceDirectory* class instantiates a *RDResource* class. It represents the */rd* CoAP resource in which CoAP servers performs resources registration, managed by the *handlePOST* method. The diagram is shown in Figure 12.

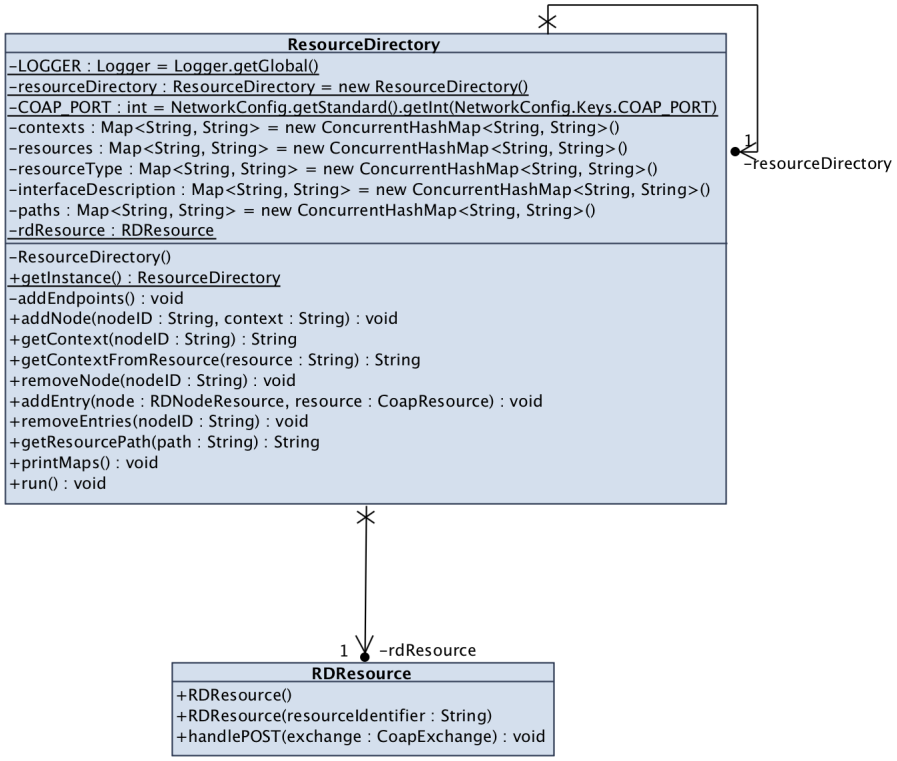


Figure 12: Class Diagram: Resource Directory

Every time a CoAP server registers a resource, its location is instantiated in the Resource Directory as a *RDNodeResource* class (Figure 13). It stores the node information and allows endpoints update and removal. The *ExpiryTask* class represents the timer task and it ensures resources deletion when their time expires.

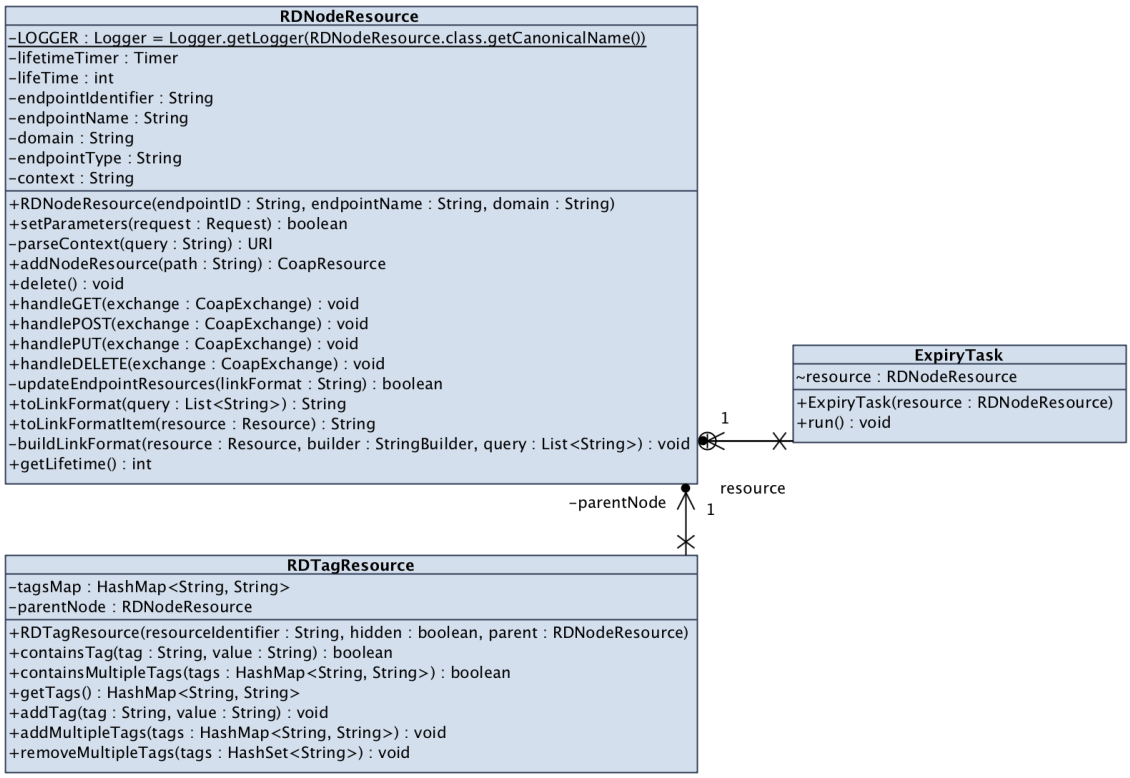


Figure 13: Class Diagram: RD Node Resource

5.3.3 *CoAP Proxy*

The *CoAPProxy* class is the element that interacts with the CoAP network. It implements the methods that allow to send request messages to the CoAP servers, and to register and unregister to a resource.

The observing service is managed by the *ObserverThread*, which listens for resource notifications and returns them to the proxy.

The proxy also implements the caching service. The cache is represented by the *ProxyCacheResource* class, which contains all the stored data in the form of *CacheResource* classes. Each cache entry is identified by a *CacheKey*.

The class diagram is shown in Figure 14.

5.3.4 *AJ Object Manager*

The *AJObjectManagerApp* is the component that deals with the AllJoyn devices. It implements the methods used in object creation, object cancellation and notifications registration. The class contains all the registered CoAP resources in the form of *CoAPResource* AllJoyn objects, that implement the *CoAPInterface* interface. The last one provides the methods that allow the client to handle with CoAP resources (GET, POST, DELETE, and observing service).

The *CoAPResource* class also contains the resource information, like its path, the resource type and the interface description.

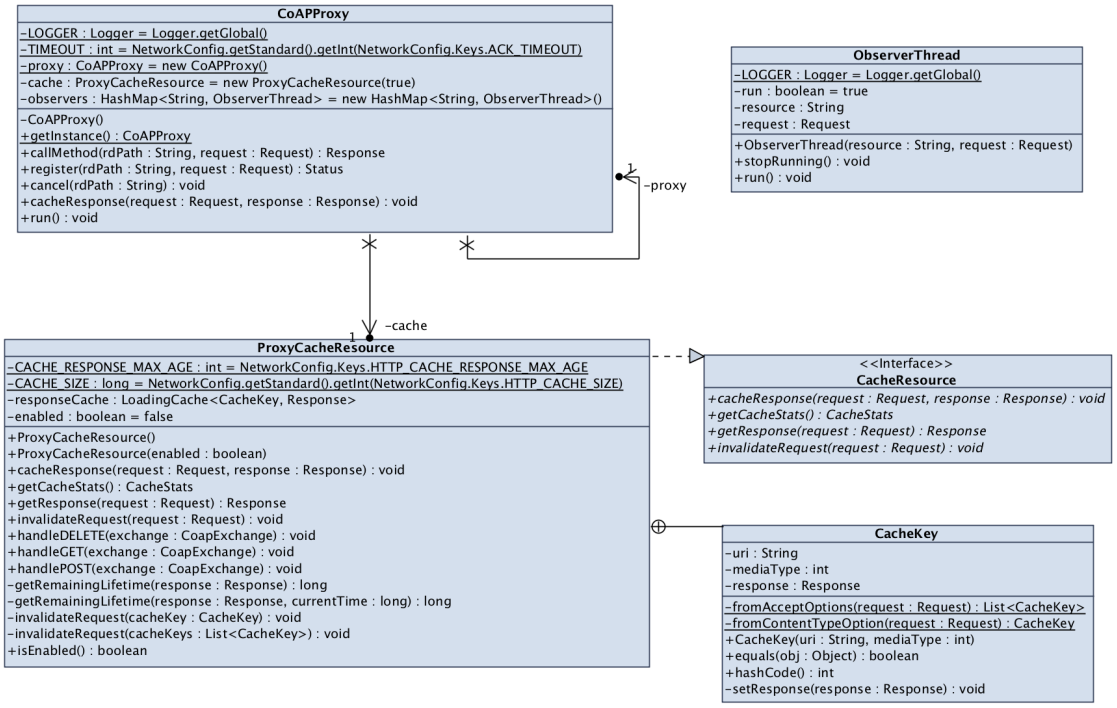


Figure 14: Class Diagram: CoAP Proxy

The Object Manager uses the *BridgeAboutData* to inform the AllJoyn network about the application information and the advertised AllJoyn objects.

The Figure 15 shows the Object Manager class diagram.

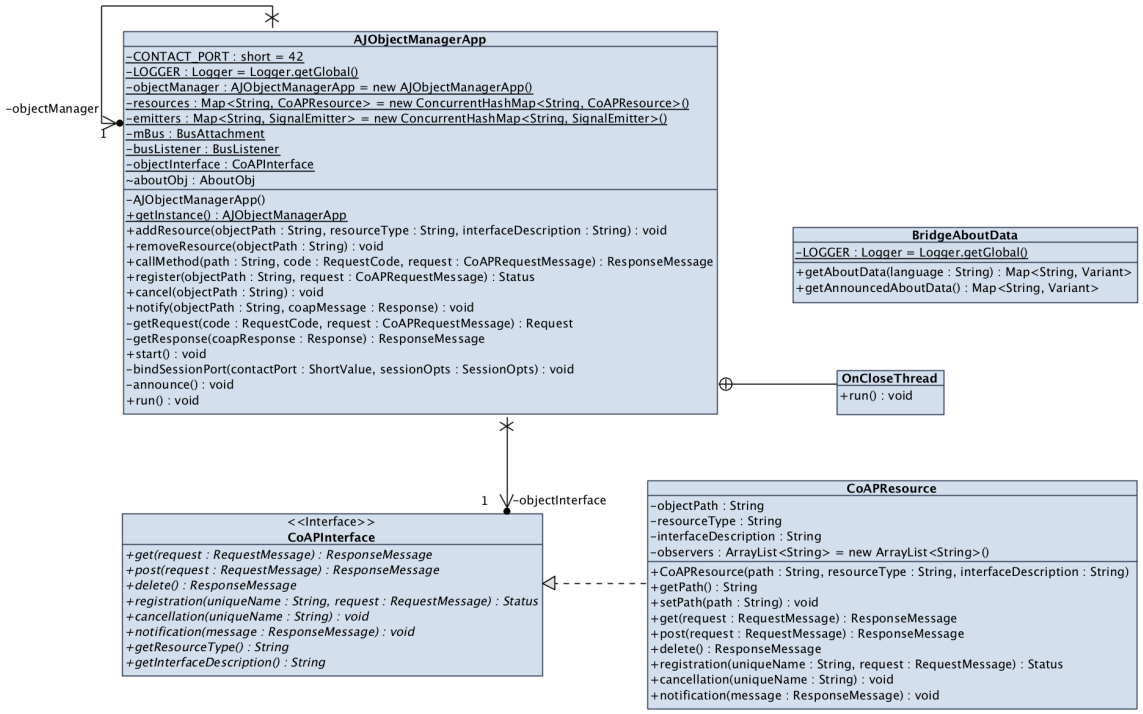


Figure 15: Class Diagram: Object Manager

The Object Manager interacts with the AllJoyn client using an AllJoyn representation of CoAP messages. The request and the response messages are mapped, respectively, into

the *RequestMessage* class and the *ResponseMessage* class, which implement the *CoAPRequestMessage* and the *CoAPResponseMessage* interfaces (Figure 16).

The messages classes contain the attributes that could be useful to the AllJoyn client to interact with the CoAP resources. The request message allows to set the message payload and the query string; the response message has the response code and the message payload. Both the request and the response messages contain the *Options* field.

The *RequestCode* and the *ResponseCode* enumerators are provided by the *CoAP* class.

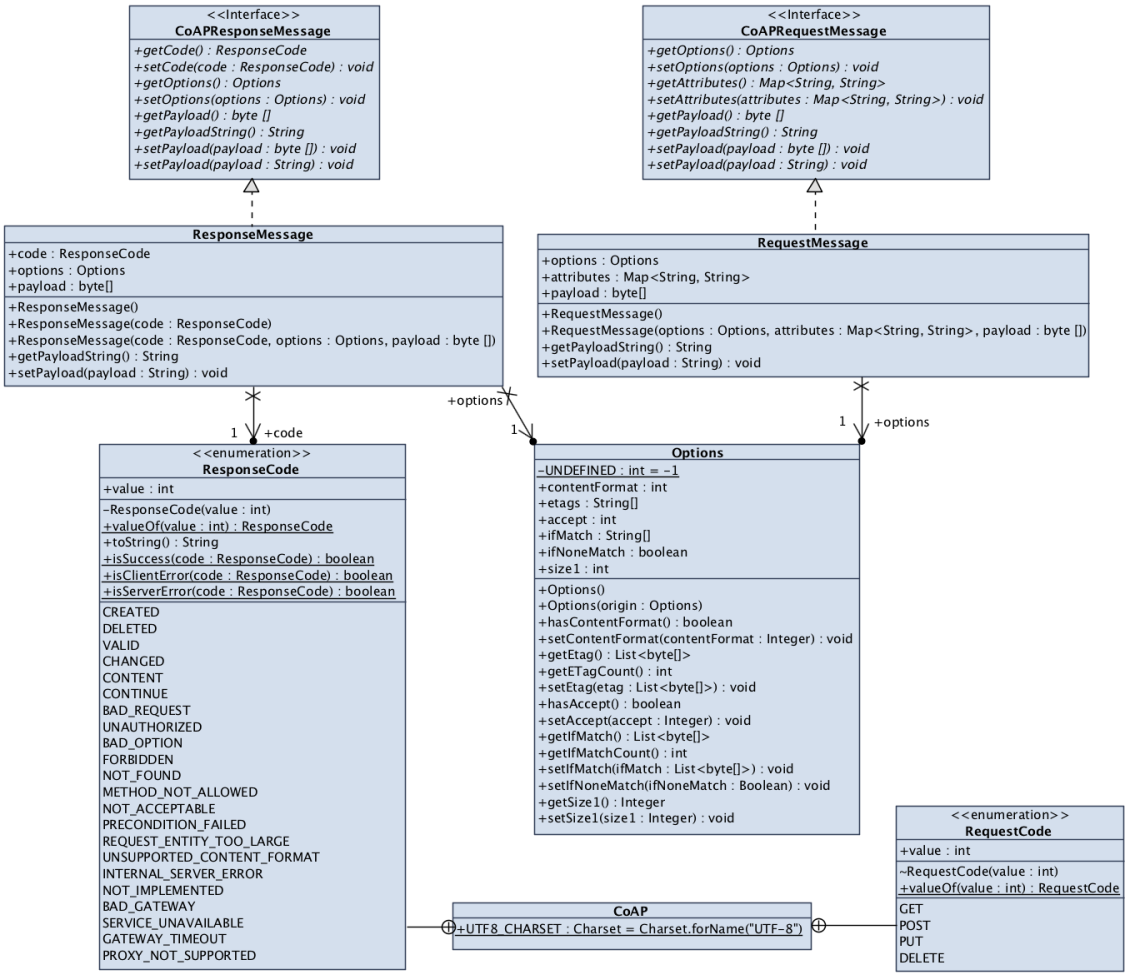


Figure 16: Class Diagram: Request and Response Messages

5.4 SEQUENCE DIAGRAMS

5.4.1 *Resource registration*

Registration: "Created"

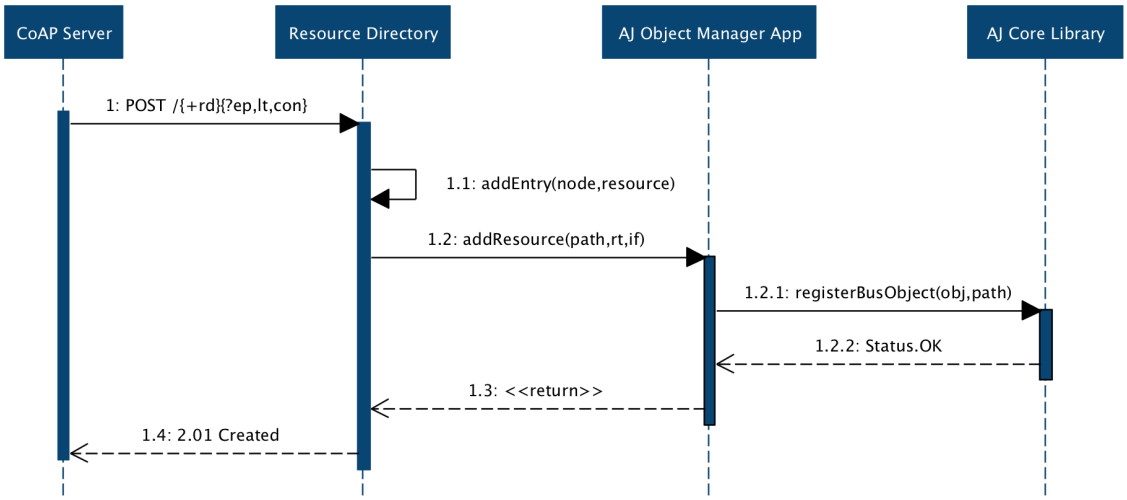


Figure 17: Sequence Diagram: Registration

The sequence diagram in Figure 17 shows the interaction between the components during the registration phase, when the request has been fulfilled and resulted in a just created new location:

1. The *CoAP Server* registers a resource sending a POST method to the *Resource Directory*.

1.1. The *Resource Directory* adds a new entry including the new registered resource, its associated node, and assigns it a timer initialized to *lt* (or to the default value, if the option is not present).

1.2. The *Resource Directory* sends the resource path and its attributes to the *AllJoyn Object Manager Application*.

1.2.1. The *AJ Object Manager Application* creates the new objects and registers them with their object path in the *AllJoyn Core Library*.

1.2.2. If errors do not occur, the *AJ Core Library* responds to the *AJ Object Manager App* with a *Status.OK*.

1.3. The *AJ Object Manager App* informs the *Resource Directory* that it has terminated.

1.4. If errors do not occur, the *Resource Directory* sends a response to the *CoAP Server* with response code 2.01 and the resource location in the message payload.

Registration: "Bad Request"

Figure 18 shows the sequence diagram with a bad request during the registration phase. It happens when the request cannot be understood by the RD due to malformed syntax.

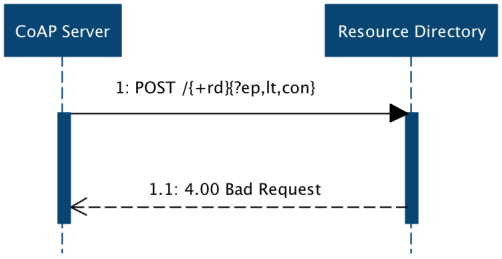


Figure 18: Sequence Diagram: Registration "Bad Request"

1. The *CoAP Server* sends a POST method to the *Resource Directory* with incorrect syntax.

1.1. The *Resource Directory* sends a response message to the *CoAP Server* with response code 4.00.

5.4.2 *Resource update*

Update: "Changed"

The sequence diagram in Figure 19 shows the interaction between the components during a resource update, when a resource exists at the request URI and the enclosed representation should be considered as a modified version of that resource. The update interface is used by an endpoint to refresh or update its registration with an RD. To use the interface, the endpoint sends a POST request to the resource returned in the Location option in the response to the first registration. An update may update the lifetime or context parameters if they have changed since the last registration or update.

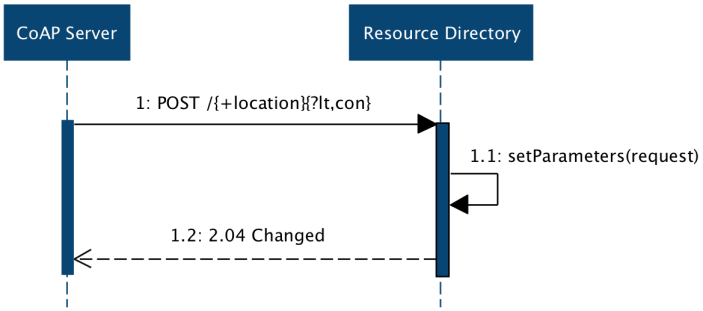


Figure 19: Sequence Diagram: Update

1. The *CoAP Server* sends a POST method to the *Resource Directory* including its *location* in the URI.

1.1. The *Resource Directory* updates the entries corresponding to the received location.

1.2. If errors do not occur, the *Resource Directory* sends a response to the *CoAP Server* with response code 2.04.

Update: "Bad Request"

Figure 20 shows the sequence diagram with a bad request during a resource update. It means that the request could not be understood by the RD due to malformed syntax.

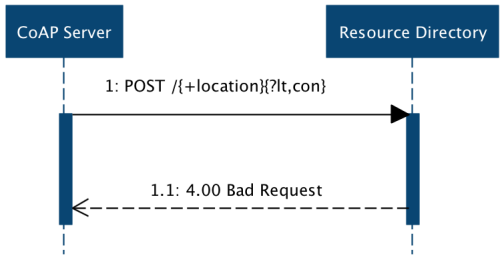


Figure 20: Sequence Diagram: Update "Bad Request"

1. The *CoAP Server* sends a POST method to the *Resource Directory* with incorrect syntax.

1.1. The *Resource Directory* sends a response message to the *CoAP Server* with response code 4.00.

Update: "Not Found"

Figure 21 shows the sequence diagram in the case in which the resource to be updated is not found or it has expired. It means that the RD has not found anything matching the request URI.

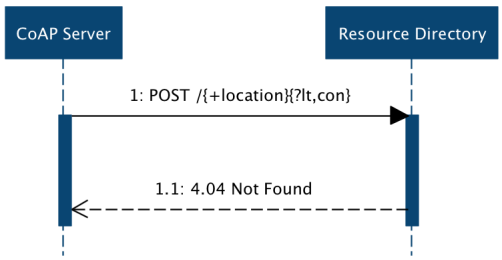


Figure 21: Sequence Diagram: Update "Not Found"

1. The *CoAP Server* sends a POST method to the *Resource Directory*.

1.1. The *Resource Directory* does not find a match with the request URI and it sends a response message to the *CoAP Server* with response code 4.04.

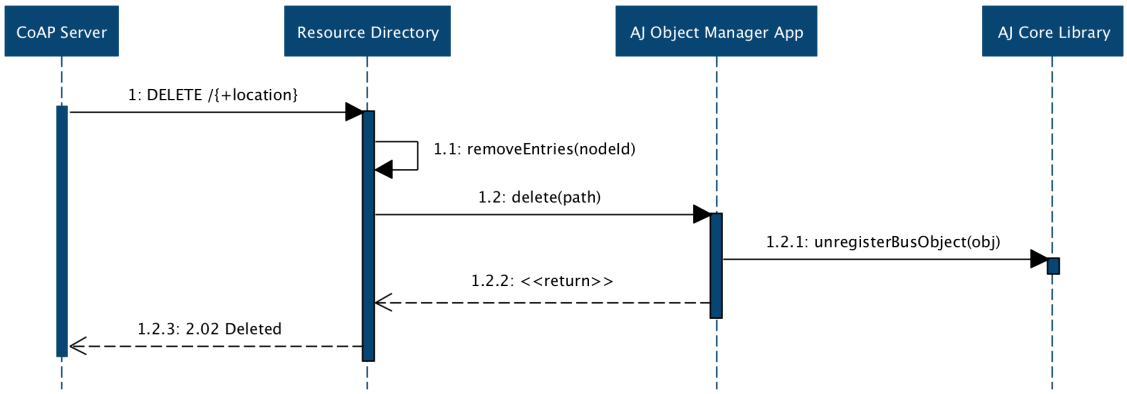


Figure 22: Sequence Diagram: Resource Removal

5.4.3 *Resource removal*

Removal: "Deleted"

The sequence diagram in Figure 22 shows the interaction between the components during the resource removal:

1. The *CoAP Server* sends a DELETE method to the *Resource Directory* specifying the *location* to be removed.

1.1. The *Resource Directory* removes the entries corresponding to the specified *location*.

1.2. The *Resource Directory* informs the *AJ Object Manager Application* that some resources have been removed.

1.2.1. The *AJ Object Manager App* unregisters the objects corresponding to the removed location from the *AJ Core Library*.

1.2.2. The *AJ Object Manager App* informs the *Resource Directory* that it is completed.

1.2.3. The *Resource Directory* sends a response to the *CoAP Server* with response code 2.02.

Removal: "Bad Request"

Figure 23 shows the sequence diagram with a bad request during a resource removal. It means that the request could not be understood by the RD due to malformed syntax.

1. The *CoAP Server* sends a DELETE method to the *Resource Directory* with incorrect syntax.

1.1. The *Resource Directory* sends a response message to the *CoAP Server* with response code 4.00.

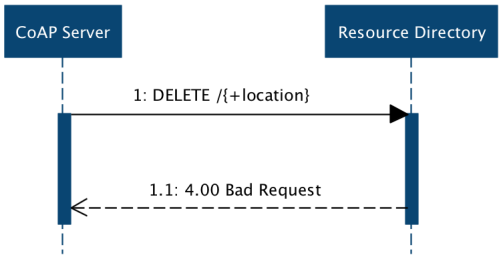


Figure 23: Sequence Diagram: Removal "Bad Request"

Removal: "Not Found"

Figure 24 shows the sequence diagram in the case in which the resource to be deleted is not found or it has expired. It means that the RD has not found anything matching the request URI.

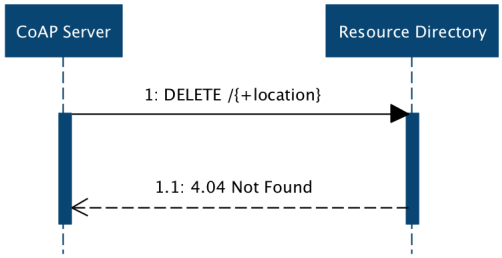


Figure 24: Sequence Diagram: Removal "Not Found"

1. The *CoAP Server* sends a DELETE method to the *Resource Directory*.

1.1. The *Resource Directory* does not find a match with the request URI and it sends a response message to the *CoAP Server* with response code 4.04.

5.4.4 *GET method call*

GET: no cached data

The message exchange that occours during a GET method call is described in the Sequence Diagram in Figure 25. In this case, the *Cache* does not contain stored data for the request.

1. The *AJ App* executes a *get* method call on the AJ Proxy Object.

1.1. The client *AJ Core Library* sends a METHOD_CALL message to the Bridge *AJ Core Library*.

1.1.1. The Bridge *AJ Core Library* calls a *get* method call on the CoAP resource object in the *AJ Object Manager App*.

1.1.1.1. The *AJ Object Manager App* makes a request message and adds the GET request code to it.

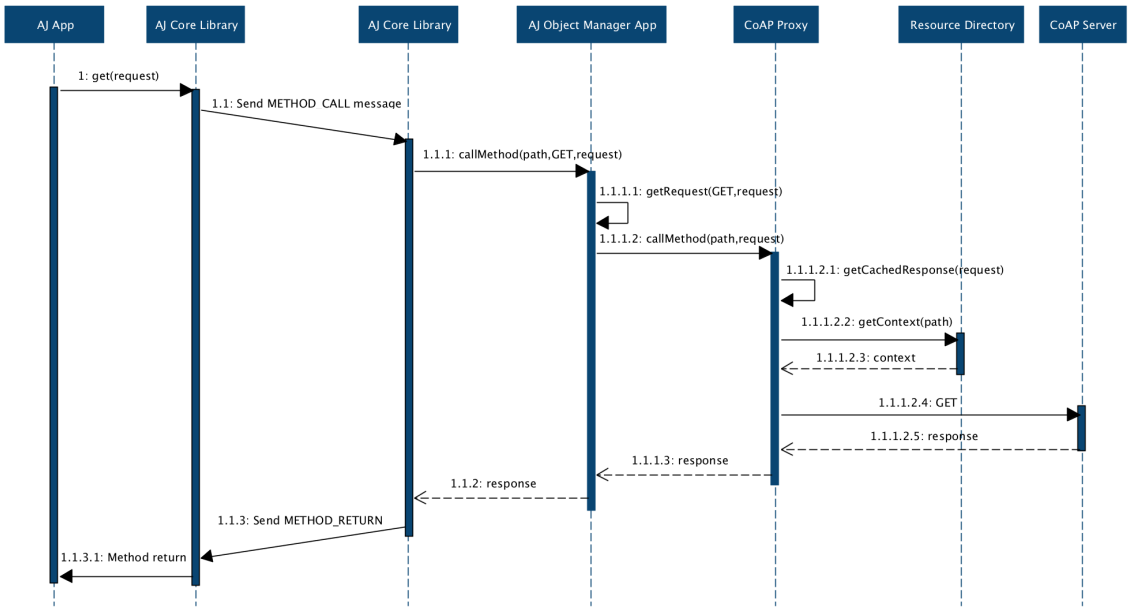


Figure 25: Sequence Diagram: GET Method without cached data

1.1.1.2. The *AJ Object Manager App* sends the request message to the *CoAP Proxy*.

1.1.1.2.1. The *CoAP Proxy* checks on the *Cache* to find stored data for the received request.

1.1.1.2.2. The *CoAP Proxy* asks the *Resource Directory* for the destination context.

1.1.1.2.3. The *Resource Directory* gives back the destination IP address and port.

1.1.1.2.4. The *CoAP Proxy* sends the request message to the *CoAP Server*.

1.1.1.2.5. The *CoAP Server* sends the response message to the *CoAP Proxy*. The response message can contain one of the following response codes:

- *2.05 Content*: the request is fulfilled and the returned payload in the response is a representation of the target resource.
- *4.xx Client Error*: the client seems to have made a mistake.
- *5.xx Server Error*: the server is aware that it has made a mistake or is incapable of performing the request.

1.1.1.3. The *CoAP Proxy* returns the response message to the *AJ Object Manager App*.

1.1.2. The *AJ Object Manager* sends the response to the *AJ Core Library*.

1.1.3. The Bridge *AJ Core Library* sends a METHOD_RETURN message to the client *AJ Core Library*.

1.1.3.1. The client *AJ Core Library* sends a method return to the *AllJoyn App*.

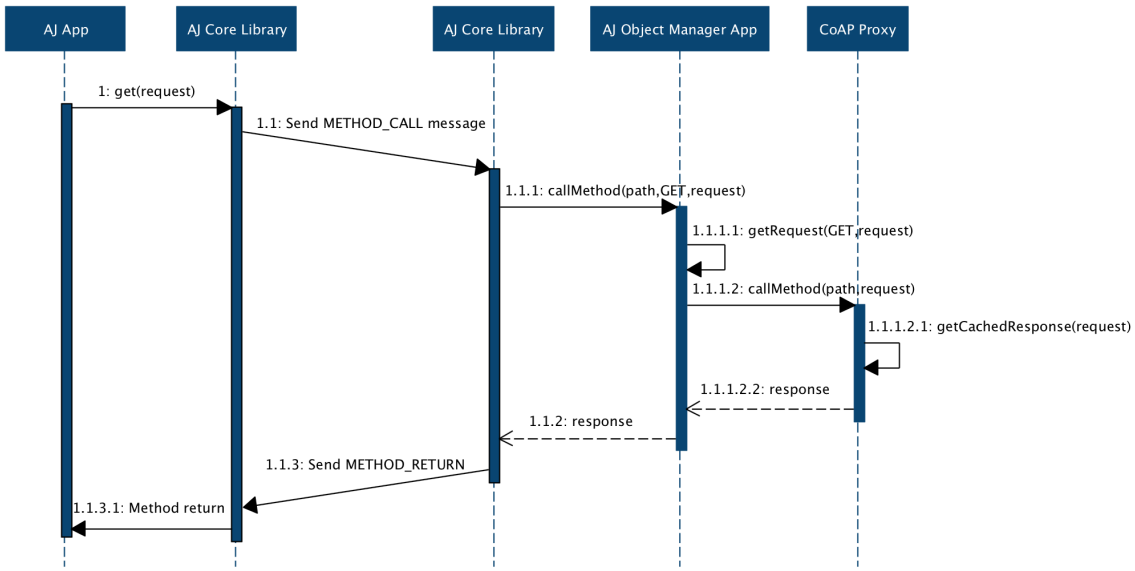


Figure 26: Sequence Diagram: GET Method with cached data

GET: cached data

The message exchange that occours during a GET method call is described in the Sequence Diagram in Figure 26. In this case, the *Cache* contains stored data for the request.

1. The *AJ App* executes a *get* method call on the AJ Proxy Object.

1.1. The client *AJ Core Library* sends a METHOD_CALL message to the Bridge *AJ Core Library*.

1.1.1. The Bridge *AJ Core Library* sends a *get* method call on the CoAP resource in the *AJ Object Manager App*.

1.1.1.1. The *AJ Object Manager App* makes a request message and adds the GET request code to it.

1.1.1.2. The *AJ Object Manager App* sends the request message to the *CoAP Proxy*.

1.1.1.2.1. The *CoAP Proxy* checks on the *Cache* to find stored data for the received request.

1.1.1.2.2. The *CoAP Proxy* has stored data inside its *Cache* and returns it to the *AJ Object Manager App*.

1.1.2. The *AJ Object Manager App* sends a method response to the *AJ Core Library*.

1.1.3. The Bridge *AJ Core Library* sends a METHOD_RETURN message to the client *AJ Core Library*.

1.1.3.1. The client *AJ Core Library* sends a method return to the *AllJoyn App*.

5.4.5 *POST method call*

The message exchange that occours during a POST method call is described in the Sequence Diagram in Figure 27.

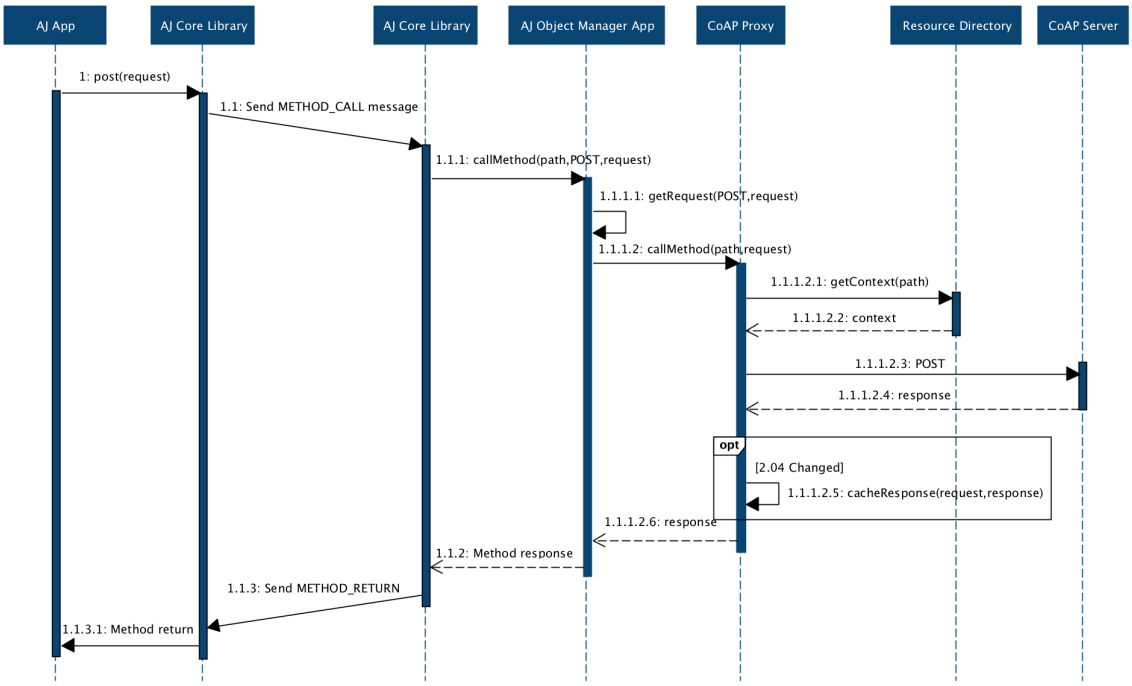


Figure 27: Sequence Diagram: POST Method

1. The *AJ App* executes a *post* method call on the AJ Proxy Object.

1.1. The client *AJ Core Library* sends a METHOD_CALL message to the Bridge *AJ Core Library*.

1.1.1. The Bridge *AJ Core Library* sends a *post* method call to the CoAP resource in the *AJ Object Manager App*.

1.1.1.1. The *AJ Object Manager App* makes a request message and adds the POST request code to it.

1.1.1.2. The *AJ Object Manager App* sends the request message to the *CoAP Proxy*.

1.1.1.2.1. The *CoAP Proxy* asks the *Resource Directory* for the destination context.

1.1.1.2.2. The *Resource Directory* gives back the destination IP address and port.

1.1.1.2.3. The *CoAP Proxy* sends the request message to the *CoAP Server*.

1.1.1.2.4. The *CoAP Server* sends the response message to the *CoAP Proxy*. The response message can contain one of the following response codes:

- *2.04 Changed*: the request is fulfilled and the returned payload in the response, if any, is a representation of the action result.

- *4.xx Client Error*: the client seems to have made a mistake.
- *5.xx Server Error*: the server is aware that it has made a mistake or is incapable of performing the request.

1.1.1.2.5. If the message response code is the *2.04 Changed* code, the *CoAP Proxy* updates the *Cache* entries corresponding to the updated resources to the new values.

1.1.1.2.6. The *CoAP Proxy* returns the response message to the *AJ Object Manager App*.

1.1.2. The *AJ Object Manager* sends a method response to the *AJ Core Library*.

1.1.3. The Bridge *AJ Core Library* sends a METHOD_RETURN message to the client *AJ Core Library*.

1.1.3.1. The client *AJ Core Library* sends a method return to the *AllJoyn App*.

5.4.6 *DELETE method call*

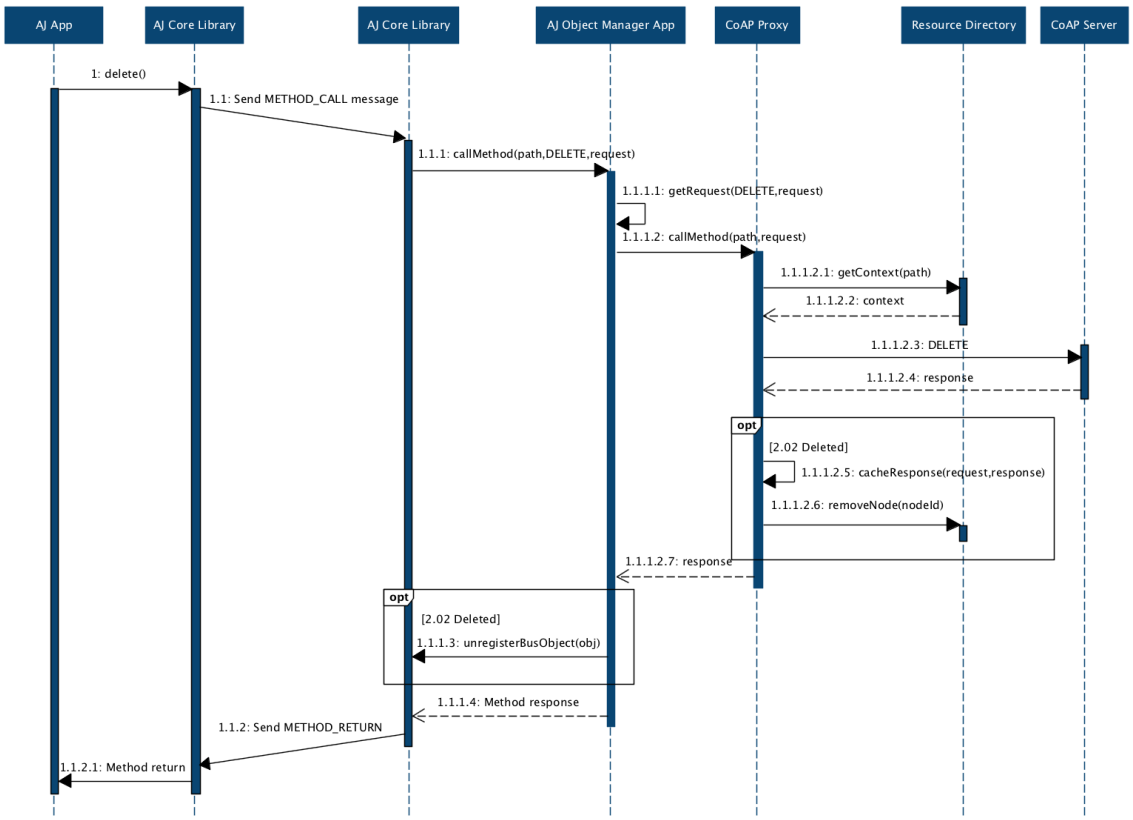


Figure 28: Sequence Diagram: DELETE Method

The message exchange that occours during a DELETE method call is described in the Sequence Diagram in Figure 28.

1. The *AJ App* executes a *delete* method call on the AJ Proxy Object.

1.1. The client *AJ Core Library* sends a METHOD_CALL message to the Bridge *AJ Core Library*.

1.1.1. The Bridge *AJ Core Library* sends a *delete* method call to the *AJ Object Manager App*.

1.1.1.1. The *AJ Object Manager App* makes a request message and adds the DELETE request code to it.

1.1.1.2. The *AJ Object Manager App* sends the request message to the *CoAP Proxy*.

1.1.1.2.1. The *CoAP Proxy* asks the *Resource Directory* for the destination context.

1.1.1.2.2. The *Resource Directory* gives back the destination IP address and port.

1.1.1.2.3. The *CoAP Proxy* sends the request message to the *CoAP Server*.

1.1.1.2.4. The *CoAP Server* sends the response message to the *CoAP Proxy*. The response message can contain one of the following response codes:

- *2.02 Deleted*: the request is fulfilled, the resource ceases to be available and the returned payload in the response, if any, is a representation of the action result.
- *4.xx Client Error*: the client seems to have made a mistake.
- *5.xx Server Error*: the server is aware that it has made a mistake or is incapable of performing the request.

1.1.1.2.5. If the message response code is the *2.02 Deleted* code, the *CoAP Proxy* removes the *Cache* entries corresponding to the deleted resources.

1.1.1.2.6. If the message response code is the *2.02 Deleted* code, the *CoAP Proxy* tells the *Resource Directory* to remove the entries corresponding to the deleted resources.

1.1.1.2.7. The *CoAP Proxy* returns the response message to the *AJ Object Manager App*.

1.1.1.3. If the message response code is the *2.02 Deleted* code, the *AJ Object Manager App* unregisters the objects corresponding to the deleted resources.

1.1.1.4. The *AJ Object Manager* sends a method response to the *AJ Core Library*.

1.1.2. The Bridge *AJ Core Library* sends a METHOD_RETURN message to the client *AJ Core Library*.

1.1.2.1. The client *AJ Core Library* sends a method return to the *AllJoyn App*.

5.4.7 *Observing registration*

The message exchange that occours during a observing registration is described in the Sequence Diagram in Figure 29.

1. The *AJ Application* registers a signal handler for the notification it wants to receive.
2. The *AJ App* executes a *registration* method call on the AJ Proxy Object in which it also includes its unique name, in order to be recognized by the bridge.

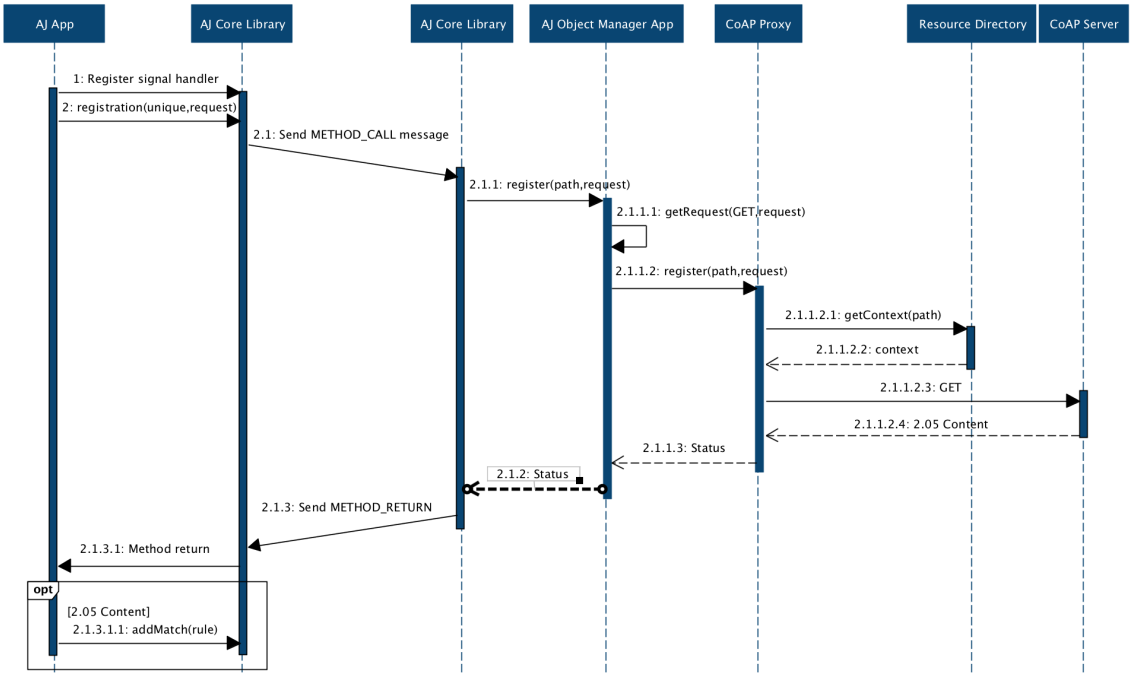


Figure 29: Sequence Diagram: Observing Registration

2.1. The client *AJ Core Library* sends a METHOD_CALL message to the Bridge *AJ Core Library*.

2.1.1. The Bridge *AJ Core Library* sends a *registration* method call to the *AJ Object Manager App*.

2.1.1.1. The *AJ Object Manager App* makes an extended GET request message with *observe* option to 0 (register).

2.1.1.2. The *AJ Object Manager App* sends the request message to the *CoAP Proxy*.

2.1.1.2.1. The *CoAP Proxy* asks the *Resource Directory* for the destination context.

2.1.1.2.2. The *Resource Directory* gives back the destination IP address and port.

2.1.1.2.3. The *CoAP Proxy* sends the request message to the *CoAP Server*.

2.1.1.2.4. The *CoAP Server* sends the response message to the *CoAP Proxy*. The response message can contain one of the following response codes:

- *2.05 Content*: the request is fulfilled and the returned payload in the response is a representation of the target resource.
- *4.xx Client Error*: the client seems to have made a mistake.
- *5.xx Server Error*: the server is aware that it has made a mistake or is incapable of performing the request.

2.1.1.3. The *CoAP Proxy* returns the status to the *AJ Object Manager App*.

2.1.2. The *AJ Object Manager* sends the status as the method response to the *AJ Core Library*.

2.1.3. The Bridge *AJ Core Library* sends a METHOD_RETURN message to the client *AJ Core Library*.

2.1.3.1. The client *AJ Core Library* sends a method return to the *AllJoyn App*.

2.1.3.1.1. If the message response code is the *2.05 Content* code, the *AJ App* adds a match rule for the signal it wants to receive.

5.4.8 *Notification*

The sequence diagram in Figure 30 shows the interaction between the components when there is a notification to be sended. Notifications typically have a 2.05 (*Content*) response code and a payload in the same Content-Format as the initial response.

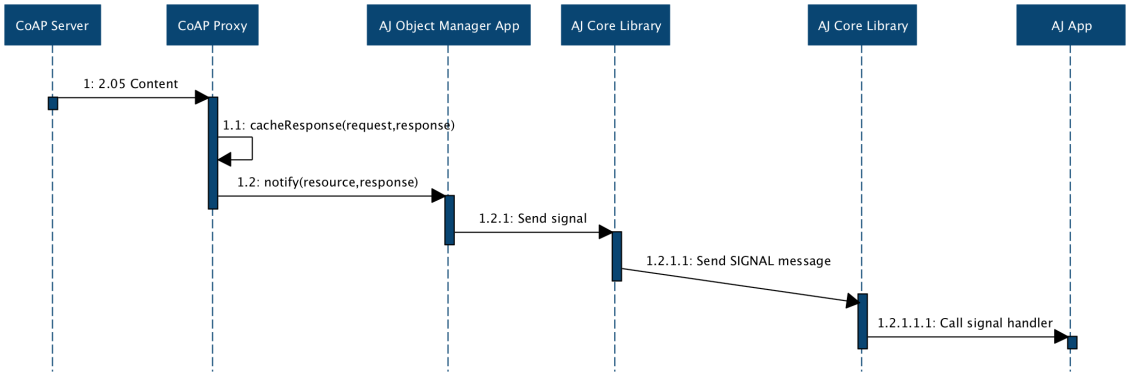


Figure 30: Sequence Diagram: Notification

1. The *CoAP Server* sends a notification message to the *CoAP Proxy*.

1.1. The *CoAP Proxy* updates the stored data corresponding to the received resource.

1.2. The *CoAP Proxy* informs the *AJ Object Manager App* that a new notification has been arrived and passes it the data.

1.2.1. The *AJ Object Manager* sends a signal to the *AJ Core Library*.

1.2.1.1. The Bridge *AJ Core Library* sends a SIGNAL message to the client *AJ Core Library*.

1.2.1.1.1. The client *AJ Core Library* calls the signal handler corresponding to the received signal.

5.4.9 *Observing cancellation*

The sequence diagram in Figure 31 shows the interaction between the components when a client wants to be unsubscribed from the observing service.

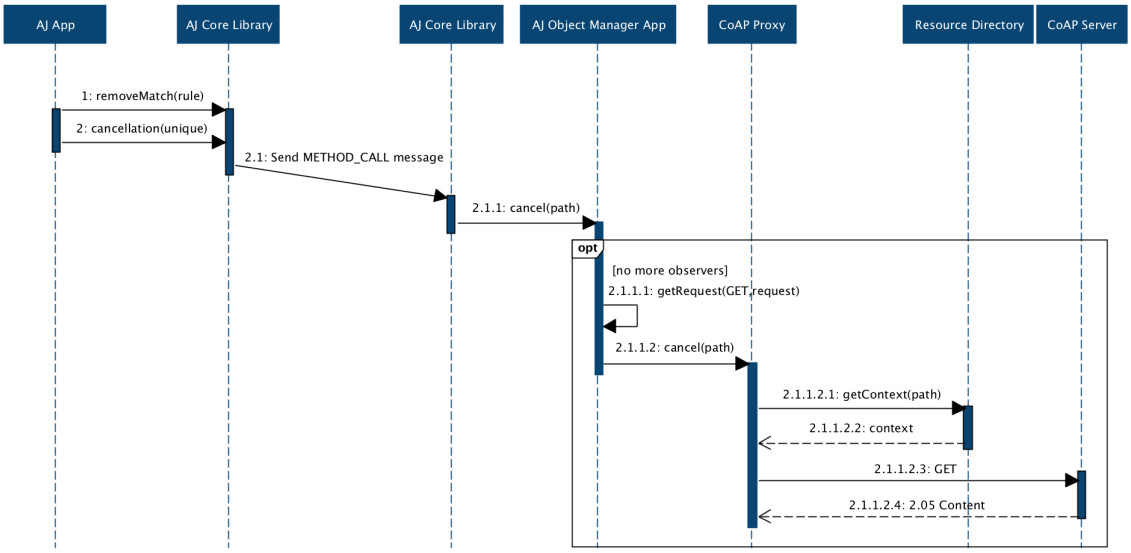


Figure 31: Sequence Diagram: Observing Cancellation

1. The *AllJoyn Application* removes the match rule for the signal it does not want to receive anymore.

2. The *AJ App* calls a *cancellation* method on the AJ Proxy Object including its unique name, in order to be recognized by the bridge.

2.1. The client *AJ Core Library* sends a METHOD_CALL message to the Bridge *AJ Core Library*.

2.1.1. The Bridge *AJ Core Library* sends a *cancellation* method call to the *AJ Object Manager App*.

The *AJ Object Manager App* decreases the number of observers for that resource. If no more clients want to receive notification from a resource, then the Bridge is no more interested to receive notification from the CoAP Server.

A client (the Bridge, in this case) that is no longer interested in receiving notification for a resource can simply forget the observation. When the server sends the next notification, the client will not recognize the token in the message and thus will return a *Reset* message. This cause the server to remove the associated entry from the list of observers.

In some circumstaces, it may be desiderable to cancel an observation and release the resources allocated by the server to it more eagerly. In this case, a client can explicity deregister by issuing a GET request that has the *Token* field set to the token of the observation to be cancelled and includes an *Observe* option with the value of 1 (deregister):

2.1.1.1. The *AJ Object Manager App* makes an extended GET request message and includes an *observe* option with the value set to 1 (deregister).

2.1.1.2. The *AJ Object Manager App* sends the request message to the *CoAP Proxy*.

2.1.1.2.1. The *CoAP Proxy* asks the *Resource Directory* for the destination context.

2.1.1.2.2. The *Resource Directory* gives back the destination IP address and port.

2.1.1.2.3. The *CoAP Proxy* sends the request message to the *CoAP Server*.

2.1.1.2.4. The *CoAP Server* sends the response message to the *CoAP Proxy*.

6 TEST REPORT

In order to examine the proper functioning of the system some tests were made. They cover all the system use cases.

This chapter is the test report in which the setup, the specification, the summary and the result for each test we made are specified.

6.1 ENVIRONMENT SETUP

6.1.1 *Requirements*

The following are required in order to execute these test cases:

- An AllJoyn-enabled device on which the CoAP Bridge will run.
- A device on which the CoAP server will run.
- An AllJoyn-enabled device on which the AllJoyn client will run.
- A Wi-Fi access point (referred to as the personal AP).

During the test cases, the three listed applications will run on the same machine.

6.1.2 *Preconditions*

Before running these test cases, it is assumed that:

- The device under test (or DUT) is connected to the personal AP.
- At least one process on the DUT is announcing its capabilities through its About announcement, including its support for the CoAP interface.
- The CoAP server application implements at least one CoAP resource.
- The CoAP server application knows the Bridge Resource Directory IP address and port.

6.2 TEST CASE SPECIFICAION

6.2.1 *BRIDGE-TC-01 CoAP Resources Registration*

Objectives

Verify that CoAP devices can register their resources to the Bridge Resource Directory.

Procedure

1. The CoAP test device sends a GET request to *"well-known/core"* to the Bridge Resource Directory and includes a Resource Type parameter with the value *"core.rd"* in the query string.
2. After receiving the response, the CoAP test device sends a POST request to the Bridge Resource Directory and includes the Endpoint Name parameter in the query string and a list of resources in the payload.
3. The CoAP test device waits to receive the response message.

Expected results

- The CoAP test device sends a GET request to *"well-known/core"* and receives a response message with *2.05* as response code and a link format entry for the Resource Directory in the payload.
- The CoAP test device sends a POST request to the Bridge Resource Directory and receives a response message with *2.01* response code and a location in the payload.
  - If not equal to *2.01*, the test fails.
  - If no response messages arrive, the test fails.

6.2.2 *BRIDGE-TC-02 CoAP Resources Cancellation*

Objectives

Verify that CoAP devices can unregister their previously registered resources from the Bridge Resource Directory.

Procedure

1. The CoAP test device sends a GET request to *"well-known/core"* to the Bridge Resource Directory and includes a Resource Type parameter with the value *"core.rd"* in the query string.
2. After receiving the response, the CoAP test device sends a POST request to the Bridge Resource Directory and includes the Endpoint Name parameter in the query string and a list of resources in the payload.

3. The CoAP test device waits to receive the response message.
4. The CoAP test device sends a DELETE request to the Bridge Resource Directory and includes its location in the request URI.
5. The CoAP test device waits to receive the response message.

Expected results

- The CoAP test device sends a GET request to *"well-known/core"* and receives a response message with *2.05* as response code and a link format entry for the Resource Directory in the payload.
- The CoAP test device sends a POST request to the Bridge Resource Directory and receives a response message with *2.01* response code and a location in the payload.
- The CoAP test device sends a DELETE request to the Bridge Resource Directory and receives a response message with *2.02* response code.
  - If not equal to *2.02*, the test fails.
  - If no response messages arrive, the test fails.

6.2.3 *BRIDGE-TC-03 Discovery of AllJoyn Objects*

Objectives

Verify that AllJoyn devices can discovery the objects advertised by the Bridge.

Procedure

1. The AllJoyn test device calls the *findAdvertisedName()* method with *"com.bridge.coap"* as parameter.
2. After that the *foundAdvertisedName()* method has been called, the AllJoyn test device joins a session with the Bridge.
3. The AllJoyn test device calls the *whoImplements()* method with *"com.bridge.Coap"* as parameter.
4. The AllJoyn test device listens for an About announcement from the Bridge.

Expected results

- The AllJoyn test device receives a *foundAdvertisedName()* method call.
- The AllJoyn test device joins a session with the Bridge.
- The AllJoyn test device receives an About announcement from the Bridge containing all its advertised objects that implement the *"com.bridge.Coap"* interface.

6.2.4 *BRIDGE-TC-04 GET Method Call*

Objectives

Verify that AllJoyn devices can call the *get()* method to call the method of the CoAP device through the Bridge. The test starts after that the session with the Bridge has been established and the test device has received an About announcement.

Procedure

1. The AllJoyn test device creates an AllJoyn Proxy Object for the object announced by the Bridge.
2. The AllJoyn test device calls the *get()* method on the Proxy Object with the request message as parameter.
3. The AllJoyn test device waits to receive the response message as method return.

Expected results

- The AllJoyn test device calls the *get()* method.
- The Bridge sends a GET request method call to the CoAP server in which the resource is and the method returns a response message with a valid response code.
- The AllJoyn test device receives a response message as return of the *get()* method and the response message contains a valid response code.
  - If a bus exception occurs, the test fails.

6.2.5 *BRIDGE-TC-05 GET Method Call with Cached Response*

Objectives

Verify that if AllJoyn devices call the *get()* method on the same resource with the same request more than once, the Bridge gives it a cached response. The test starts after that the session with the Bridge has been established and the test device has received an About announcement.

Procedure

1. The AllJoyn test device creates an AllJoyn Proxy Object for the object announced by the Bridge.
2. The AllJoyn test device calls the *get()* method on the Proxy Object with the request message as parameter.
3. The AllJoyn test device waits to receive the response message as method return.

4. The AllJoyn test device calls the *get()* method on the Proxy Object with the same request message as parameter.
5. The AllJoyn test device waits to receive the response message as method return.

Expected results

- The AllJoyn test device calls the *get()* method.
- The Bridge sends a GET request method call to the CoAP server in which the resource is and the method returns a response message with a valid response code.
- The AllJoyn test device receives a response message as return of the *get()* method and the response message contains a valid response code.
- The AllJoyn test device calls the *get()* method with the same request.
- The Bridge gets the response message in the cache and sends it to the test device.
- The AllJoyn test device receives a response message as return of the *get()* method and the response message contains a valid response code.
  - If a bus exception occurs, the test fails.

6.2.6 *BRIDGE-TC-06 POST Method Call*

Objectives

Verify that AllJoyn devices can call the *post()* method to call the method of the CoAP device through the Bridge. The test starts after that the session with the Bridge has been established and the test device has received an About announcement.

Procedure

1. The AllJoyn test device creates an AllJoyn Proxy Object for the object announced by the Bridge.
2. The AllJoyn test device calls the *post()* method on the Proxy Object with the request message as parameter.
3. The AllJoyn test device waits to receive the response message as method return.

Expected results

- The AllJoyn test device calls the *post()* method.
- The Bridge sends a POST request method call to the CoAP server in which the resource is and the method returns a response message with a valid response code.

- The AllJoyn test device receives a response message as return of the *post()* method and the response message contains a valid response code.
  - If a bus exception occurs, the test fails.

6.2.7 *BRIDGE-TC-07 DELETE Method Call*

Objectives

Verify that AllJoyn devices can call the *delete()* method to call the method of the CoAP device through the Bridge. The test starts after that the session with the Bridge has been established and the test device has received an About announcement.

Procedure

1. The AllJoyn test device creates an AllJoyn Proxy Object for the object announced by the Bridge.
2. The AllJoyn test device calls the *delete()* method on the Proxy Object with the request message as parameter.
3. The AllJoyn test device waits to receive the response message as method return.

Expected results

- The AllJoyn test device calls the *delete()* method.
- The Bridge sends a DELETE request method call to the CoAP server in which the resource is and the method returns a response message with a valid response code.
- The AllJoyn test device receives a response message as return of the *delete()* method and the response message contains a valid response code.
  - If a bus exception occurs, the test fails.

6.2.8 *BRIDGE-TC-08 Observing Registration*

Objectives

Verify that AllJoyn devices can call the *registration()* method to start observing a CoAP resource. The test starts after that the session with the Bridge has been established and the test device has received an About announcement.

Procedure

1. The AllJoyn test device registers a signal handler.
2. The AllJoyn test device creates an AllJoyn Proxy Object for the object announced by the Bridge.

3. The AllJoyn test device calls the *registration()* method on the Proxy Object with its unique id and the request message as parameters.
4. The AllJoyn test device waits to receive the method return.
5. The AllJoyn test device adds a match rule for the object it has registered.

Expected results

- The AllJoyn test device successfully registers the signal handler.
- The AllJoyn test device calls the *registration()* method.
- The Bridge sends a GET request method call to the CoAP server in which the resource is and the method returns a response message with a valid response code and the observe option set to 0.
- The status code returned by the *registration()* method equals *OK*.
  - If a bus exception occurs, the test fails.
  - If not equal to *OK* and not equal to *NOT_IMPLEMENTED*, the test fails.
  - If the returned status code equals *NOT_IMPLEMENTED*, the CoAP resources cannot be observed.
- The AllJoyn test device successfully adds the signal match rule.

6.2.9 *BRIDGE-TC-09 Observing Cancellation*

Objectives

Verify that AllJoyn devices can call the *cancellation()* method to stop observing a CoAP resource. The test starts after that the session with the Bridge has been established and the test device has received an About announcement.

Procedure

1. The AllJoyn test device registers a signal handler.
2. The AllJoyn test device creates an AllJoyn Proxy Object for the object announced by the Bridge.
3. The AllJoyn test device calls the *registration()* method on the Proxy Object with its unique id and the request message as parameters.
4. The AllJoyn test device waits to receive the method return.
5. The AllJoyn test device adds a match rule for the object it has registered.
6. The AllJoyn test device calls the *cancellation()* method on the Proxy Object with its unique id as parameters.

7. The AllJoyn test device removes the match rule for the object it doesn't want more to observe.

Expected results

- The AllJoyn test device successfully registers the signal handler.
- The AllJoyn test device calls the *registration()* method.
- The Bridge sends a GET request method call to the CoAP server in which the resource is and the method returns a response message with a valid response code and the observe option set to 0.
- The status code returned by the *registration()* method equals *OK*.
- The AllJoyn test device successfully adds the signal match rule.
- The AllJoyn test device calls the *cancellation()* method.
- The Bridge sends a GET request method call to the CoAP server in which the resource is and the method returns a response message with a valid response code and the observe option set to 1.
- The AllJoyn test device successfully removes the match rule.
  - If a bus exception occurs, the test fails.

6.2.10 *BRIDGE-TC-10 Notification Arrival*

Objectives

Verify that AllJoyn devices receive signal when a CoAP resource they are following sends a notification. The test starts after that the CoAP test device has registered its resources, the session with the Bridge has been established and the test device has received an About announcement.

Procedure

1. The AllJoyn test device registers a signal handler.
2. The AllJoyn test device creates an AllJoyn Proxy Object for the object announced by the Bridge.
3. The AllJoyn test device calls the *registration()* method on the Proxy Object with its unique id and the request message as parameters.
4. The AllJoyn test device waits to receive the method return.
5. The AllJoyn test device adds a match rule for the object it has registered.

6. The CoAP test device sends two notifications to the Bridge with 5 seconds of period between them.
7. The AllJoyn test device receives notification signals and handles them.
8. The AllJoyn test device calls the *cancellation()* method on the Proxy Object with its unique id as parameters.
9. The AllJoyn test device removes the match rule for the object it doesn't want more to observe.

Expected results

- The AllJoyn test device successfully registers the signal handler.
- The AllJoyn test device calls the *registration()* method.
- The Bridge sends a GET request method call to the CoAP server in which the resource is and the method returns a response message with a valid response code and the observe option set to 0.
- The status code returned by the *registration()* method equals *OK*.
- The AllJoyn test device successfully adds the signal match rule.
- The CoAP test device sends two notifications to the Bridge.
- The Bridge sends two signals to the AllJoyn side for the object representing the resource that sent the notifications.
- The AllJoyn test device receives two notification signals and successfully handles them.
- The AllJoyn test device calls the *cancellation()* method.
- The Bridge sends a GET request method call to the CoAP server in which the resource is and the method returns a response message with a valid response code and the observe option set to 1.
- The AllJoyn test device successfully removes the match rule.
  - If a bus exception occurs, the test fails.

6.3 TEST CASES SUMMARY

6.3.1 *BRIDGE-TC-01 CoAP Resources Registration*

VALIDATION TEST RESULT			
Test Case ID	BRIDGE-TC-01	**Test Title**	CoAP Resources Registration
Test Setup			
The CoAP server emulates two resources: a light sensor and a temperature sensor. In order to register them, it sends a POST request message to the bridge in which it specifies the URI, the content format, the resource type and the interface description for both the resources. In the message URI query, the CoAP server includes the endpoint name field, unique within the CoAP network.			
Result			
OK			
Execution Info			
Date: 02/05/2016 Author: David Costa			

6.3.2 *BRIDGE-TC-02 CoAP Resources Cancellation*

VALIDATION TEST RESULT			
Test Case ID	BRIDGE-TC-02	**Test Title**	CoAP Resources Cancellation
Test Setup			
The CoAP server emulates two resources: a light sensor and a temperature sensor. In order to register them, it sends a POST request message to the bridge in which it specifies the URI, the content format, the resource type and the interface description for both the resources. In the message URI query, the CoAP server includes the endpoint name field, unique within the CoAP network.			
Result			
OK			
Execution Info			
Date: 02/05/2016 Author: David Costa			

6.3.3 *BRIDGE-TC-03 Discovery of AllJoyn Objects*

VALIDATION TEST RESULT			
Test Case ID	BRIDGE-TC-03	**Test Title**	Discovery of AllJoyn Objects
Test Setup			
CoAP resources are previously registered on the bridge. The AllJoyn client looks for the available CoAP objects in the network. It means that the parameter of the whoimplement() function is set to ”com.bridge.Coap”.			
Result			
OK			
Execution Info			
Date: 02/05/2016 Author: David Costa			

6.3.4 *BRIDGE-TC-04 GET Method Call*

VALIDATION TEST RESULT			
Test Case ID	BRIDGE-TC-04	**Test Title**	GET Method Call
Test Setup			
CoAP resources are previously registered. CoAP server emulates the temperature sensor with the temperature set to 18 degrees. During the execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, and then he calls the GET method with the accept field set to 40.			
Result			
OK			
Execution Info			
Date: 02/05/2016 Author: David Costa			

6.3.5 *BRIDGE-TC-05 GET Method Call with Cached Response*

VALIDATION TEST RESULT			
Test Case ID	BRIDGE-TC-05	**Test Title**	GET Method Call with Cached Response
Test Setup			
CoAP resources are previously registered. CoAP server emulates the temperature sensor with the temperature set to 18 degrees. During the execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, and then he calls the GET method with the accept field set to 40.			
Result			
OK			
Execution Info			
Date: 02/05/2016 Author: David Costa			

6.3.6 *BRIDGE-TC-06 POST Method Call*

VALIDATION TEST RESULT			
Test Case ID	BRIDGE-TC-06	**Test Title**	POST Method Call
Test Setup			
CoAP resources are previously registered. During execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, and then he calls the POST method with the content format field set to 40 and the payload field set to 22.			
Result			
OK			
Execution Info			
Date: 02/05/2016 Author: David Costa			

6.3.7 *BRIDGE-TC-07 DELETE Method Call*

VALIDATION TEST RESULT			
Test Case ID	BRIDGE-TC-07	**Test Title**	DELETE Method Call
Test Setup			
CoAP resources are previously registered. During execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, and then he calls the DELETE method.			
Result			
OK			
Execution Info			
Date: 02/05/2016 Author: David Costa			

6.3.8 *BRIDGE-TC-08 Observing Registration*

VALIDATION TEST RESULT			
Test Case ID	BRIDGE-TC-08	**Test Title**	Observing Registration
Test Setup			
CoAP resources are previously registered as observable resources. During execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, and then he calls the registration method with the accept field set to 40.			
Result			
OK			
Execution Info			
Date: 02/05/2016 Author: David Costa			

6.3.9 *BRIDGE-TC-09 Observing Cancellation*

VALIDATION TEST RESULT			
Test Case ID	BRIDGE-TC-09	**Test Title**	Observing Cancellation
Test Setup			
CoAP resources are previously registered as observable resources. During execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, calls the registration method with the accept field set to 40, and then he calls the cancellation method.			
Result			
OK			
Execution Info			
Date: 02/05/2016 Author: David Costa			

6.3.10 *BRIDGE-TC-10 Notification Arrival*

VALIDATION TEST RESULT			
Test Case ID	BRIDGE-TC-10	**Test Title**	Notification Arrival
Test Setup			
CoAP resources are previously registered as observable resources. The resources are configured to set a notification every 5 seconds. CoAP server emulates the temperature sensor with the temperature set to 18 degrees. During execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, calls the registration method with the accept field set to 40, he waits for the arrival of two notifications, and then he calls the cancellation method.			
Result			
OK			
Execution Info			
Date: 02/05/2016 Author: David Costa			

6.4 DETAILED TEST RESULT

6.4.1 *BRIDGE-TC-01 CoAP Resource Registration*

Test setup

The CoAP server emulates two resources: a light sensor and a temperature sensor. In order to register them, it sends a POST request message to the bridge in which it specifies the URI, the content format, the resource type and the interface description for both the resources. In the message URI query, the CoAP server includes the endpoint name field, unique within the CoAP network.

```
Uri-query: ep=node1
Payload:
</temp>;ct=40;rt=temperature;if=sensor,
</light>;ct=40;rt=light;if=sensor
```

Test results

In this section the selection of the output logs lines are shown as an evidence of the test result.
Bridge Console:

```
mag 02, 2016 12:28:22 PM org.eclipse.californium.core.network.config.NetworkConfig
    createStandardWithFile
INFO: Loading standard properties from file Californium.properties
mag 02, 2016 12:28:22 PM org.eclipse.californium.core.CoapServer start
INFO: Starting server
mag 02, 2016 12:28:22 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:642e:d9ff:fe8a:9043%awdl0:5683
mag 02, 2016 12:28:22 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:a65e:60ff:fece:6c25%en0:5683
mag 02, 2016 12:28:22 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /131.114.221.215:5683
mag 02, 2016 12:28:22 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:0:0:0:1%lo0:5683
mag 02, 2016 12:28:22 PM it.dc.bridge.rd.ResourceDirectory run
INFO: ResourceDirectory listening on port 5683
mag 02, 2016 12:28:22 PM it.dc.bridge.om.AJObjectManagerApp start
INFO: Starting AllJoyn server
mag 02, 2016 12:28:23 PM it.dc.bridge.om.AJObjectManagerApp start
INFO: BusAttachment.connect successful on null
mag 02, 2016 12:28:23 PM it.dc.bridge.om.AJObjectManagerApp start
INFO: BusAttachment.request 'com.bridge.coap' successful
mag 02, 2016 12:28:23 PM it.dc.bridge.om.AJObjectManagerApp start
INFO: BusAttachment.advertiseName 'com.bridge.coap' successful
mag 02, 2016 12:28:23 PM it.dc.bridge.om.AJObjectManagerApp bindSessionPort
INFO: BusAttachment.bindSessionPort successful
mag 02, 2016 12:28:23 PM it.dc.bridge.om.AJObjectManagerApp start
INFO: Announce called announcing SessionPort: 42
mag 02, 2016 12:28:25 PM it.dc.bridge.rd.RDResource handlePOST
INFO: Registration request: /131.114.221.215
mag 02, 2016 12:28:25 PM it.dc.bridge.om.AJObjectManagerApp addResource
INFO: Registered bus object: /rd/2080/temp
mag 02, 2016 12:28:25 PM it.dc.bridge.om.AJObjectManagerApp announce
INFO: Announce called announcing SessionPort: 42
mag 02, 2016 12:28:25 PM it.dc.bridge.om.AJObjectManagerApp addResource
INFO: Registered bus object: /rd/2080/light
mag 02, 2016 12:28:25 PM it.dc.bridge.om.AJObjectManagerApp announce
INFO: Announce called announcing SessionPort: 42
mag 02, 2016 12:28:25 PM it.dc.bridge.rd.RDResource handlePOST
INFO: Adding new endpoint: coap://131.114.221.215:5682
```

CoAP Server Console:

```
mag 02, 2016 12:28:25 PM org.eclipse.californium.core.network.config.NetworkConfig
    createStandardWithFile
INFO: Loading standard properties from file Californium.properties
```

```
mag 02, 2016 12:28:25 PM dc.coaptest.ServerTest setup
INFO:
Start ServerTest
mag 02, 2016 12:28:25 PM org.eclipse.californium.core.CoapServer start
INFO: Starting server
mag 02, 2016 12:28:25 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:642e:d9ff:fe8a:9043%awdl0:5682
mag 02, 2016 12:28:25 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:a65e:60ff:fece:6c25%en0:5682
mag 02, 2016 12:28:25 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /131.114.221.215:5682
mag 02, 2016 12:28:25 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:0:0:0:1%lo0:5682
mag 02, 2016 12:28:25 PM dc.coaptest.ServerTest rdDiscovery
INFO: Send GET request for RD discovery
mag 02, 2016 12:28:25 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at 0.0.0.0/0.0.0.0:0
mag 02, 2016 12:28:25 PM org.eclipse.californium.core.network.EndpointManager createDefaultEndpoint
INFO: Created implicit default endpoint 0.0.0.0/0.0.0.0:63193
mag 02, 2016 12:28:25 PM dc.coaptest.ServerTest rdDiscovery
INFO: Received response with code:2.05
mag 02, 2016 12:28:25 PM dc.coaptest.ServerTest rdRegistration
INFO: Send POST request for resource registration
mag 02, 2016 12:28:25 PM dc.coaptest.ServerTest rdRegistration
INFO: Received response with code:2.01
mag 02, 2016 12:28:25 PM dc.coaptest.ServerTest rdRegistration
INFO: Received location:/rd/2080
```

Encountered problems

None.

Deviations from test case

None.

6.4.2 *BRIDGE-TC-02 CoAP Resource Cancellation*

Test setup

The CoAP server emulates two resources: a light sensor and a temperature sensor. In order to register them, it sends a POST request message to the bridge in which it specifies the URI, the content format, the resource type and the interface description for both the resources. In the message URI query, the CoAP server includes the endpoint name field, unique within the CoAP network.

Test results

In this section the selection of the output logs lines are shown as an evidence of the test result.
Bridge Console:

```
mag 02, 2016 12:41:34 PM org.eclipse.californium.core.network.config.NetworkConfig
    createStandardWithFile
INFO: Loading standard properties from file Californium.properties
mag 02, 2016 12:41:34 PM org.eclipse.californium.core.CoapServer start
INFO: Starting server
mag 02, 2016 12:41:34 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:642e:d9ff:fe8a:9043%awdl0:5683
mag 02, 2016 12:41:34 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:a65e:60ff:fece:6c25%en0:5683
mag 02, 2016 12:41:34 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /131.114.221.215:5683
mag 02, 2016 12:41:34 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:0:0:0:1%lo0:5683
mag 02, 2016 12:41:34 PM it.dc.bridge.rd.ResourceDirectory run
INFO: ResourceDirectory listening on port 5683
mag 02, 2016 12:41:34 PM it.dc.bridge.om.AJObjectManagerApp start
INFO: Starting AllJoyn server
mag 02, 2016 12:41:35 PM it.dc.bridge.om.AJObjectManagerApp start
INFO: BusAttachment.connect successful on null
mag 02, 2016 12:41:35 PM it.dc.bridge.om.AJObjectManagerApp start
INFO: BusAttachment.request 'com.bridge.coap' successful
mag 02, 2016 12:41:35 PM it.dc.bridge.om.AJObjectManagerApp start
INFO: BusAttachment.advertiseName 'com.bridge.coap' successful
mag 02, 2016 12:41:35 PM it.dc.bridge.om.AJObjectManagerApp bindSessionPort
INFO: BusAttachment.bindSessionPort successful
mag 02, 2016 12:41:35 PM it.dc.bridge.om.AJObjectManagerApp start
INFO: Announce called announcing SessionPort: 42
mag 02, 2016 12:41:45 PM it.dc.bridge.rd.RDResource handlePOST
INFO: Registration request: /131.114.221.215
mag 02, 2016 12:41:46 PM it.dc.bridge.om.AJObjectManagerApp addResource
INFO: Registered bus object: /rd/8517/temp
mag 02, 2016 12:41:46 PM it.dc.bridge.om.AJObjectManagerApp announce
INFO: Announce called announcing SessionPort: 42
mag 02, 2016 12:41:46 PM it.dc.bridge.om.AJObjectManagerApp addResource
INFO: Registered bus object: /rd/8517/light
mag 02, 2016 12:41:46 PM it.dc.bridge.om.AJObjectManagerApp announce
INFO: Announce called announcing SessionPort: 42
mag 02, 2016 12:41:46 PM it.dc.bridge.rd.RDResource handlePOST
INFO: Adding new endpoint: coap://131.114.221.215:5682
mag 02, 2016 12:41:46 PM it.dc.bridge.rd.RDNodeResource delete
INFO: Removing endpoint: coap://131.114.221.215:5682
mag 02, 2016 12:41:46 PM it.dc.bridge.om.AJObjectManagerApp announce
INFO: Announce called announcing SessionPort: 42
mag 02, 2016 12:41:46 PM it.dc.bridge.om.AJObjectManagerApp announce
INFO: Announce called announcing SessionPort: 42
```

CoAP Server Console:

```
mag 02, 2016 12:41:45 PM org.eclipse.californium.core.network.config.NetworkConfig
    createStandardWithFile
INFO: Loading standard properties from file Californium.properties
mag 02, 2016 12:41:45 PM dc.coaptest.ServerTest setup
INFO:
Start ServerTest
mag 02, 2016 12:41:45 PM org.eclipse.californium.core.CoapServer start
INFO: Starting server
mag 02, 2016 12:41:45 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:642e:d9ff:fe8a:9043%awdl0:5682
mag 02, 2016 12:41:45 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:a65e:60ff:fece:6c25%en0:5682
```

```
mag 02, 2016 12:41:45 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /131.114.221.215:5682
mag 02, 2016 12:41:45 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at /fe80:0:0:0:0:0:0:1%lo0:5682
mag 02, 2016 12:41:45 PM dc.coaptest.ServerTest rdDiscovery
INFO: Send GET request for RD discovery
mag 02, 2016 12:41:45 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at 0.0.0.0/0.0.0.0:0
mag 02, 2016 12:41:45 PM org.eclipse.californium.core.network.EndpointManager createDefaultEndpoint
INFO: Created implicit default endpoint 0.0.0.0/0.0.0.0:65471
mag 02, 2016 12:41:45 PM dc.coaptest.ServerTest rdDiscovery
INFO: Received response with code:2.05
mag 02, 2016 12:41:45 PM dc.coaptest.ServerTest rdRegistration
INFO: Send POST request for resource registration
mag 02, 2016 12:41:46 PM dc.coaptest.ServerTest rdRegistration
INFO: Received response with code:2.01
mag 02, 2016 12:41:46 PM dc.coaptest.ServerTest rdRegistration
INFO: Received location:/rd/8517
mag 02, 2016 12:41:46 PM dc.coaptest.ServerTest cancellation
INFO: Send DELETE request for resource cancellation
mag 02, 2016 12:41:46 PM dc.coaptest.ServerTest cancellation
INFO: Received response with code:2.02
```

Encountered problems

None.

Deviations from test case

None.

6.4.3 *BRIDGE-TC-03 Discovery of AllJoyn Objects*

Test setup

The CoAP resources are previously registered on the bridge. The AllJoyn client looks for the available CoAP objects in the network. It means that the parameter of the *whoimplement()* function is set to the regular expression *"com.bridge.Coap"*.

Test results

In this section the selection of the output logs lines are shown as an evidence of the test result.
Bridge Console:

```
mag 02, 2016 1:57:47 PM it.dc.bridge.om.AJObjectManagerApp$1 acceptSessionJoiner
INFO: SessionPortListener.acceptSessionJoiner called
mag 02, 2016 1:57:47 PM it.dc.bridge.om.AJObjectManagerApp$1 acceptSessionJoiner
INFO: SessionPortListener.acceptSessionJoiner called
mag 02, 2016 1:57:47 PM it.dc.bridge.om.AJObjectManagerApp$1 sessionJoined
INFO: SessionPortListener.sessionJoined(42, 1129795701, :WMVdnuwD.2)
mag 02, 2016 1:57:47 PM it.dc.bridge.om.AJObjectManagerApp$1 sessionJoined
INFO: SessionPortListener.sessionJoined(42, 1129795701, :WMVdnuwD.2)
```

```
mag 02, 2016 1:58:14 PM it.dc.bridge.om.AJObjectManagerApp$1 acceptSessionJoiner
INFO: SessionPortListener.acceptSessionJoiner called
mag 02, 2016 1:58:14 PM it.dc.bridge.om.AJObjectManagerApp$1 acceptSessionJoiner
INFO: SessionPortListener.acceptSessionJoiner called
mag 02, 2016 1:58:14 PM it.dc.bridge.om.AJObjectManagerApp$1 sessionJoined
INFO: SessionPortListener.sessionJoined(42, 832032745, :_j-Lfibq.2)
mag 02, 2016 1:58:14 PM it.dc.bridge.om.AJObjectManagerApp$1 sessionJoined
INFO: SessionPortListener.sessionJoined(42, 832032745, :_j-Lfibq.2)
```

AllJoyn Client Console:

```
mag 02, 2016 1:58:14 PM it.dc.ajtest.Client setup
INFO: BusAttachment.connect successful on null
mag 02, 2016 1:58:14 PM it.dc.ajtest.Client setup
INFO: BusAttachment.findAdvertisedName successful com.bridge.coap
mag 02, 2016 1:58:14 PM it.dc.ajtest.Client$MyBusListener foundAdvertisedName
INFO: BusListener.foundAdvertisedName(com.bridge.coap, 256, com.bridge.coap)
mag 02, 2016 1:58:14 PM it.dc.ajtest.Client$MyBusListener foundAdvertisedName
INFO: BusListener.foundAdvertisedName(com.bridge.coap, 4, com.bridge.coap)
mag 02, 2016 1:58:14 PM it.dc.ajtest.Client$MyBusListener foundAdvertisedName
INFO: BusAttachement.joinSession successful sessionId = 0
mag 02, 2016 1:58:14 PM it.dc.ajtest.Client$MyBusListener foundAdvertisedName
INFO: BusAttachement.joinSession successful sessionId = 832032745
mag 02, 2016 1:58:14 PM it.dc.ajtest.Client setup
INFO: BusAttachment.registerSignalHandlers successful
mag 02, 2016 1:58:15 PM it.dc.ajtest.Client findObjects
INFO: BusAttachment.whoImplements successful com.bridge.Coap
```

Screenshots

In this section the screenshot of the AllJoyn Client application is shown in Figure 32 as an evidence of the test result.

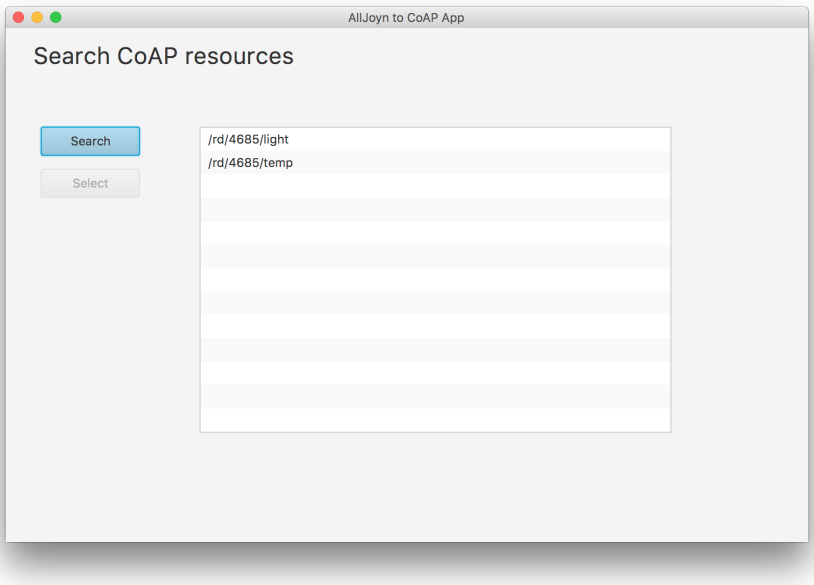


Figure 32: AllJoyn Client App: Discovery

Encountered problems

None.

Deviations from test case

None.

6.4.4 *BRIDGE-TC-04 GET Method Call*

Test setup

The CoAP resources are previously registered. CoAP server emulates the temperature sensor with the temperature set to 18 degrees. During the execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, and then he calls the GET method with the accept field set to 40.

Test results

In this section the selection of the output logs lines are shown as an evidence of the test result.
AllJoyn Client Console:

```
mag 02, 2016 2:21:35 PM it.dc.ajtest.Client get
INFO: Send GET request to:/rd/411/temp
```

Bridge Console:

```
mag 02, 2016 2:21:35 PM it.dc.bridge.om.AJObjectManagerApp callMethod
INFO: Object Manager received a GET method call on the object /rd/411/temp
mag 02, 2016 2:21:35 PM it.dc.bridge.proxy.CoAPProxy callMethod
INFO: CoAP Proxy sends a GET method call to coap://131.114.221.215:5682 on the resource /temp
mag 02, 2016 2:21:35 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at 0.0.0.0/0.0.0.0:0
mag 02, 2016 2:21:35 PM org.eclipse.californium.core.network.EndpointManager createDefaultEndpoint
INFO: Created implicit default endpoint 0.0.0.0/0.0.0.0:51259
```

CoAP Server Console:

```
mag 02, 2016 2:21:35 PM dc.coaptest.MyResource handleGET
INFO: Received GET request from /131.114.221.215 on temp
```

Screenshots

In this section the screenshot of the AllJoyn Client application is shown in Figure 33 as an evidence of the test result.

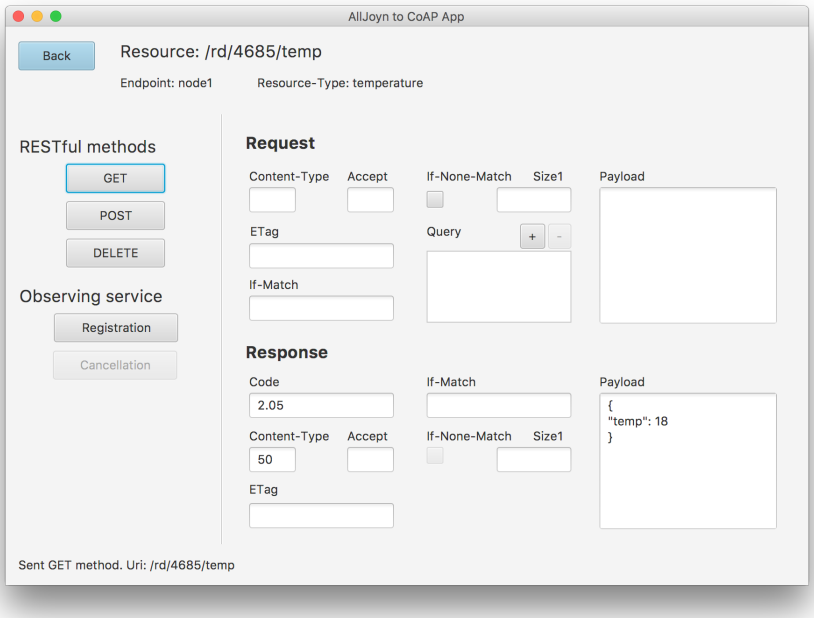


Figure 33: AllJoyn Client App: GET Method Call

Encountered problems

None.

Deviations from test case

None.

6.4.5 *BRIDGE-TC-05 GET Method Call with Cached Response*

Test setup

CoAP resources are previously registered. CoAP server emulates the temperature sensor with the temperature set to 18 degrees. During the execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, and then he calls the GET method with the accept field set to 40.

Test results

In this section the selection of the output logs lines are shown as an evidence of the test result.
AllJoyn Client Console:

```
mag 02, 2016 2:24:30 PM it.dc.ajtest.Client get
INFO: Send GET request to:/rd/1141/temp
mag 02, 2016 2:24:32 PM it.dc.ajtest.Client get
INFO: Send GET request to:/rd/1141/temp
```

Bridge Console:

```
mag 02, 2016 2:24:30 PM it.dc.bridge.om.AJObjectManagerApp callMethod
INFO: Object Manager received a GET method call on the object /rd/1141/temp
mag 02, 2016 2:24:30 PM it.dc.bridge.proxy.CoAPProxy callMethod
INFO: CoAP Proxy sends a GET method call to coap://131.114.221.215:5682 on the resource /temp
mag 02, 2016 2:24:30 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at 0.0.0.0/0.0.0.0:0
mag 02, 2016 2:24:30 PM org.eclipse.californium.core.network.EndpointManager createDefaultEndpoint
INFO: Created implicit default endpoint 0.0.0.0/0.0.0.0:59930
mag 02, 2016 2:24:32 PM it.dc.bridge.om.AJObjectManagerApp callMethod
INFO: Object Manager received a GET method call on the object /rd/1141/temp
mag 02, 2016 2:24:32 PM it.dc.bridge.proxy.ProxyCacheResource getResponse
INFO: Cache hit
mag 02, 2016 2:24:32 PM it.dc.bridge.proxy.CoAPProxy callMethod
INFO: Cache returned ACK-2.05 MID=59813, Token=ea,
     OptionSet={"Content-Format":"application/link-format", "Max-Age":59}, "18 Cel"
```

CoAP Server Console:

```
mag 02, 2016 2:24:30 PM dc.coaptest.MyResource handleGET
INFO: Received GET request from /131.114.221.215 on temp
```

Screenshots

In this section the screenshot of the AllJoyn Client application is shown in Figure 34 as an evidence of the test result.

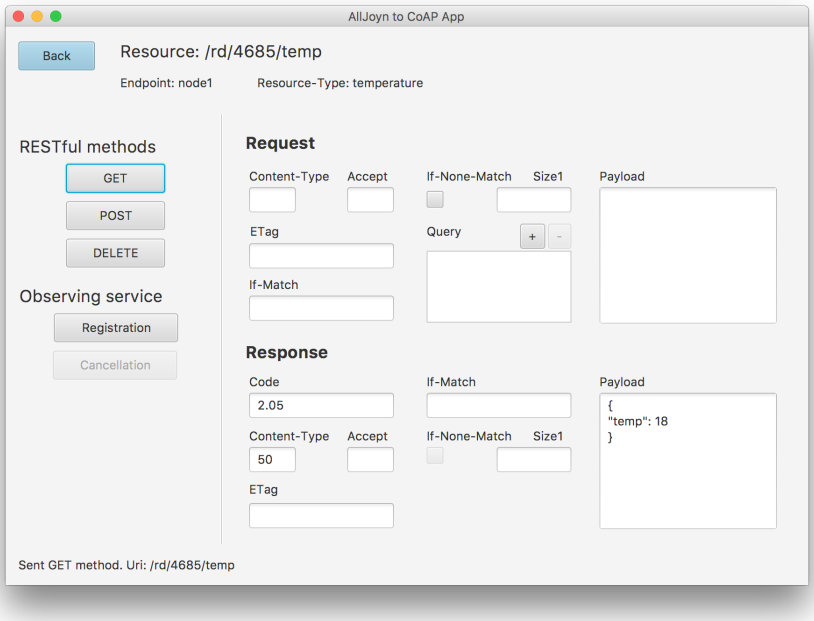


Figure 34: AllJoyn Client App: GET Method Call with Cached Response

Encountered problems

None.

Deviations from test case

None.

6.4.6 *BRIDGE-TC-06 POST Method Call*

Test setup

The CoAP resources are previously registered. During execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, and then he calls the POST method with the content format field set to 40 and the payload field set to 22.

Test results

In this section the selection of the output logs lines are shown as an evidence of the test result.
AllJoyn Client Console:

```
mag 02, 2016 2:31:14 PM it.dc.ajtest.Client post
INFO: Send POST request to:/rd/170/temp
```

Bridge Console:

```
mag 02, 2016 2:31:14 PM it.dc.bridge.om.AJObjectManagerApp callMethod
INFO: Object Manager received a POST method call on the object /rd/170/temp
mag 02, 2016 2:31:14 PM it.dc.bridge.proxy.CoAPProxy callMethod
INFO: CoAP Proxy sends a POST method call to coap://131.114.221.215:5682 on the resource /temp
mag 02, 2016 2:31:14 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at 0.0.0.0/0.0.0.0:0
mag 02, 2016 2:31:14 PM org.eclipse.californium.core.network.EndpointManager createDefaultEndpoint
INFO: Created implicit default endpoint 0.0.0.0/0.0.0.0:60039
```

CoAP Server Console:

```
mag 02, 2016 2:31:14 PM dc.coaptest.MyResource handlePOST
INFO: Received POST request from /131.114.221.215 on temp
```

Screenshots

In this section the screenshot of the AllJoyn Client application is shown in Figure 35 as an evidence of the test result.

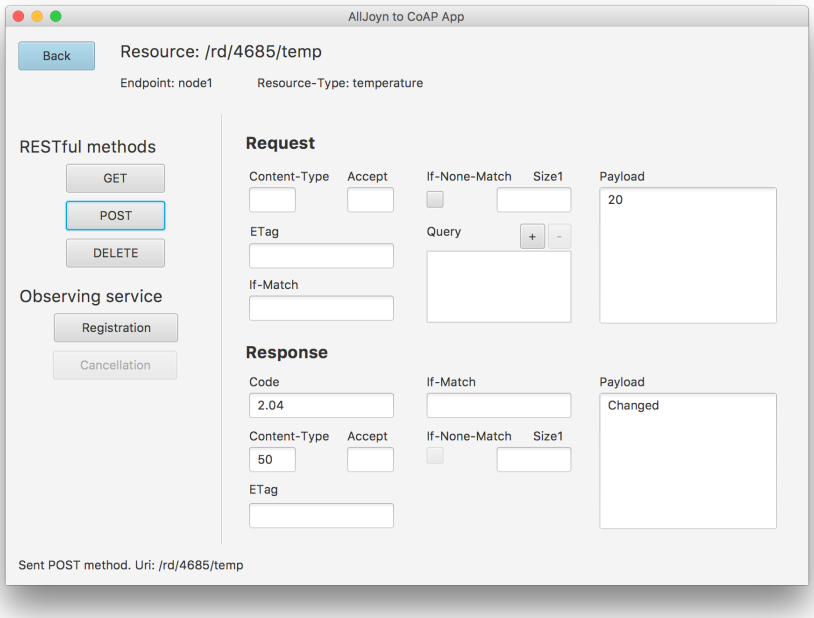


Figure 35: AllJoyn Client App: POST Method Call

Encountered problems

None.

Deviations from test case

None.

6.4.7 *BRIDGE-TC-07 DELETE Method Call*

Test setup

CoAP resources are previously registered. During execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, and then he calls the DELETE method.

Test results

In this section the selection of the output logs lines are shown as an evidence of the test result.
AllJoyn Client Console:

```
mag 02, 2016 2:33:17 PM it.dc.ajtest.Client delete
INFO: Send DELETE request to:/rd/6890/temp
```

Bridge Console:

```
mag 02, 2016 2:33:17 PM it.dc.bridge.om.AJObjectManagerApp callMethod
INFO: Object Manager received a DELETE method call on the object /rd/6890/temp
```

```
mag 02, 2016 2:33:17 PM it.dc.bridge.proxy.CoAPProxy callMethod
INFO: CoAP Proxy sends a DELETE method call to coap://131.114.221.215:5682 on the resource /temp
mag 02, 2016 2:33:17 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at 0.0.0.0/0.0.0.0:0
mag 02, 2016 2:33:17 PM org.eclipse.californium.core.network.EndpointManager createDefaultEndpoint
INFO: Created implicit default endpoint 0.0.0.0/0.0.0.0:57201
```

CoAP Server Console:

```
mag 02, 2016 2:33:17 PM dc.coaptest.MyResource handleDELETE
INFO: Received DELETE request from /131.114.221.215 on temp
```

Screenshots

In this section the screenshot of the AllJoyn Client application is shown in Figure 36 as an evidence of the test result.

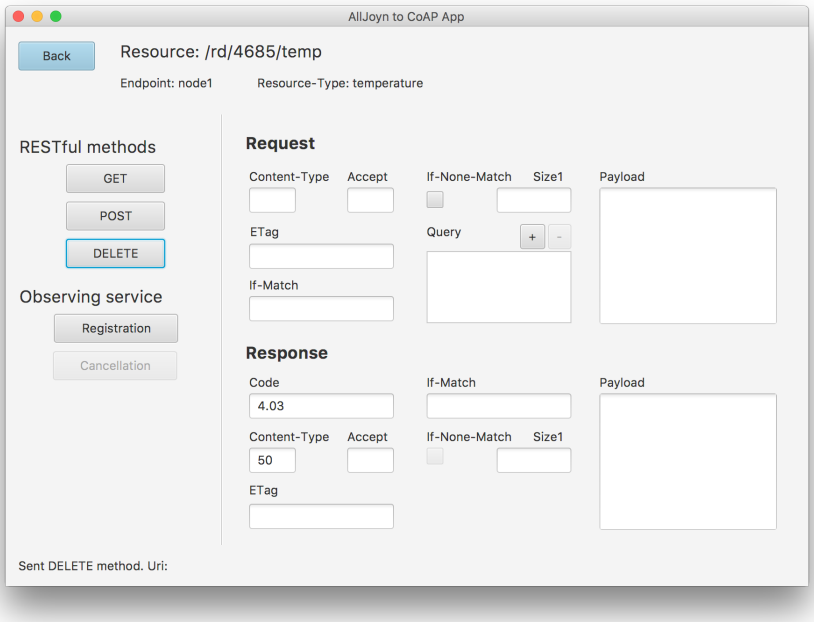


Figure 36: AllJoyn Client App: DELETE Method Call

Encountered problems

None.

Deviations from test case

None.

6.4.8 *BRIDGE-TC-o8 Observing Registration*

Test setup

The CoAP resources are previously registered as observable resources. During execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, and then he calls the registration method with the accept field set to 40.

Test results

In this section the selection of the output logs lines are shown as an evidence of the test result.
AllJoyn Client Console:

```
mag 02, 2016 2:35:56 PM it.dc.ajtest.Client register
INFO: Send registration request to:/rd/5786/temp
```

Bridge Console:

```
mag 02, 2016 2:35:56 PM it.dc.bridge.proxy.CoAPProxy register
INFO: CoAPProxy requests for observe the resource /temp from coap://131.114.221.215:5682
mag 02, 2016 2:35:56 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at 0.0.0.0/0.0.0.0:0
mag 02, 2016 2:35:56 PM org.eclipse.californium.core.network.EndpointManager createDefaultEndpoint
INFO: Created implicit default endpoint 0.0.0.0/0.0.0.0:63191
mag 02, 2016 2:35:56 PM it.dc.bridge.proxy.CoAPProxy register
INFO: Start receiving notification from coap://131.114.221.215:5682 for the resource /temp
```

CoAP Server Console:

```
mag 02, 2016 2:35:56 PM dc.coaptest.MyResource handleGET
INFO: Received GET request from /131.114.221.215 on temp
mag 02, 2016 2:35:56 PM dc.coaptest.MyResource handleGET
INFO: Received extended GET request for observing registration
mag 02, 2016 2:35:56 PM org.eclipse.californium.core.CoapResource addObserveRelation
INFO: Successfully established observe relation between /131.114.221.215:63191#9c86f1 and resource
    /temp
```

Encountered problems

None.

Deviations from test case

None.

6.4.9 *BRIDGE-TC-09 Observing Cancellation*

Test setup

The CoAP resources are previously registered as observable resources. During execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, calls the registration method with the accept field set to 40, and then he calls the cancellation method.

Test results

In this section the selection of the output logs lines are shown as an evidence of the test result.
AllJoyn Client Console:

```
mag 02, 2016 2:39:04 PM it.dc.ajtest.Client register
INFO: Send registration request to:/rd/5768/temp
mag 02, 2016 2:39:06 PM it.dc.ajtest.Client unregister
INFO: Unregister from:/rd/5768/temp
```

Bridge Console:

```
mag 02, 2016 2:39:04 PM it.dc.bridge.proxy.CoAPProxy register
INFO: CoAPProxy requests for observe the resource /temp from coap://131.114.221.215:5682
mag 02, 2016 2:39:04 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at 0.0.0.0/0.0.0.0:0
mag 02, 2016 2:39:04 PM org.eclipse.californium.core.network.EndpointManager createDefaultEndpoint
INFO: Created implicit default endpoint 0.0.0.0/0.0.0.0:60101
mag 02, 2016 2:39:04 PM it.dc.bridge.proxy.CoAPProxy register
INFO: Start receiving notification from coap://131.114.221.215:5682 for the resource /temp
mag 02, 2016 2:39:06 PM it.dc.bridge.proxy.CoAPProxy cancel
INFO: CoAPProxy requests for stop observing the resource /temp from coap://131.114.221.215:5682
mag 02, 2016 2:39:06 PM it.dc.bridge.proxy.CoAPProxy cancel
INFO: Stop receiving notification from coap://131.114.221.215:5682 for the resource /temp
```

CoAP Server Console:

```
mag 02, 2016 2:39:04 PM dc.coaptest.MyResource handleGET
INFO: Received GET request from /131.114.221.215 on temp
mag 02, 2016 2:39:04 PM dc.coaptest.MyResource handleGET
INFO: Received extended GET request for observing registration
mag 02, 2016 2:39:04 PM org.eclipse.californium.core.CoapResource addObserveRelation
INFO: Successfully established observe relation between /131.114.221.215:60101#0c32384408f1 and
    resource /temp
mag 02, 2016 2:39:06 PM dc.coaptest.MyResource handleGET
INFO: Received GET request from /131.114.221.215 on temp
mag 02, 2016 2:39:06 PM dc.coaptest.MyResource handleGET
INFO: Received extended GET request for observing cancellation
```

Encountered problems

None.

Deviations from test case

None.

6.4.10 *BRIDGE-TC-10 Notification Arrival*

Test setup

The CoAP resources are previously registered as observable resources. The resources are configured to set a notification every 5 seconds. CoAP server emulates the temperature sensor with the temperature set to 18 degrees. During execution, the user interacts with the AllJoyn client application using its GUI. He first selects the temperature resource, calls the registration method with the accept field set to 40, he waits for the arrival of two notifications, and then he calls the cancellation method.

Test results

In this section the selection of the output logs lines are shown as an evidence of the test result.
Bridge Console:

```
mag 02, 2016 2:57:24 PM it.dc.bridge.proxy.CoAPProxy register
INFO: CoAPProxy requests for observe the resource /temp from coap://131.114.221.215:5682
mag 02, 2016 2:57:24 PM org.eclipse.californium.core.network.CoapEndpoint start
INFO: Starting endpoint at 0.0.0.0/0.0.0.0:0
mag 02, 2016 2:57:24 PM org.eclipse.californium.core.network.EndpointManager createDefaultEndpoint
INFO: Created implicit default endpoint 0.0.0.0/0.0.0.0:59812
mag 02, 2016 2:57:24 PM it.dc.bridge.proxy.CoAPProxy register
INFO: Start receiving notification from coap://131.114.221.215:5682 for the resource /temp
mag 02, 2016 2:57:27 PM it.dc.bridge.om.AJObjectManagerApp notify
INFO: A notification arrived from object /rd/360/temp with code 2.05
mag 02, 2016 2:57:32 PM it.dc.bridge.om.AJObjectManagerApp notify
INFO: A notification arrived from object /rd/360/temp with code 2.05
mag 02, 2016 2:57:35 PM it.dc.bridge.proxy.CoAPProxy cancel
INFO: CoAPProxy requests for stop observing the resource /temp from coap://131.114.221.215:5682
mag 02, 2016 2:57:35 PM it.dc.bridge.proxy.CoAPProxy cancel
INFO: Stop receiving notification from coap://131.114.221.215:5682 for the resource /temp
```

AllJoyn Client Console:

```
mag 02, 2016 2:57:24 PM it.dc.ajtest.Client register
INFO: Send registration request to:/rd/360/temp
mag 02, 2016 2:57:29 PM it.dc.ajtest.Client$MySignalHandler notification
INFO: A notification arrived with code 2.05
mag 02, 2016 2:57:33 PM it.dc.ajtest.Client$MySignalHandler notification
INFO: A notification arrived with code 2.05
mag 02, 2016 2:57:35 PM it.dc.ajtest.Client unregister
INFO: Unregister from:/rd/360/temp
```

Screenshots

In this section the screenshot of the AllJoyn Client application is shown in Figure 37 as an evidence of the test result.

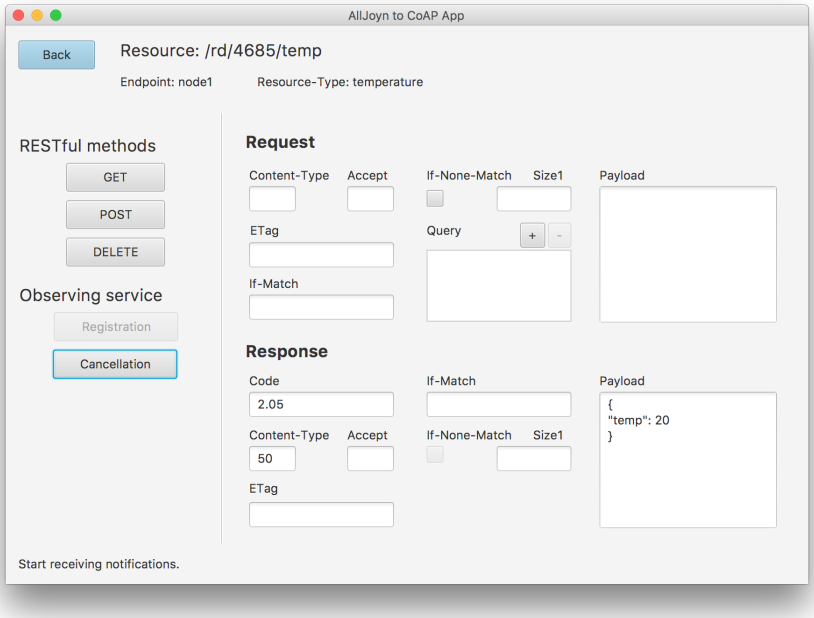


Figure 37: AllJoyn Client App: Notification Arrival

Encountered problems

None.

Deviations from test case

None.

7

TEST IN WINDOWS ENVIRONMENT

Microsoft joined the AllSeen Alliance in 2014 and added AllJoyn as a core component in Windows 10. AllJoyn capable applications can run on any of the Windows 10 devices including PCs, tablets, phones, Xbox as well as devices using Windows IoT Core.

So, due to the CoAP bridge we made, Windows devices can interact with CoAP devices natively. We have demonstrated it using the *IoT Explorer for AllJoyn* Windows application as AllJoyn client, a Wireless Sensor Network composed by several CoAP servers, and a Linux host on which the bridge runs. This chapter shows the test results.

7.1 TEST SETUP

The test case is set up in the *Dipartimento di Ingegneria dell'Informazione* of the University of Pisa, in which there is a pre-existing Wireless Sensor Network composed by 21 CoAP servers split into as many rooms in two separated floors. Figure 38 illustrates the test case plan.

(a) Downstairs

(b) Upstairs

Figure 38: CoAP Sensor Network Map

Each of these devices contains a temperature sensor, a light sensor and a humidity sensor, and exposes two resources:

- *tmp*, which provides the temperature sensor value;
- *getalldata*, which provides all the data the device exposes.

Then, the test is set up using the components as follows:

- A Windows 10 device on which the IoT Explorer for AllJoyn application will run.
- A Linux host on which the CoAP bridge will run.
- A WSN composed by devices based on CoAP, each of which provides two resources.

The bridge address and port are known by the CoAP servers, and each of them registers its resources on the bridge.

7.2 IOT EXPLORER FOR ALLJOYN

The *IoT Explorer for AllJoyn* is a Windows Universal Application for interacting with AllJoyn devices on the local proximity network [18]. Developers can list all available AllJoyn devices, inspect their interface and object structure, as well as receive signals, set and get properties, and call methods.

We used the Windows application as an AllJoyn client and we interacted with the CoAP servers through the bridge we made.

7.2.1 *Application discovery*

The IoT Explorer for AllJoyn application will search for available AllJoyn producers (devices) on the same subnet, and it will dynamically listen for AllJoyn services to announce themselves and add them to this view. Figure 39 shows the application during the discovery phase, after that the bridge application announced itself.

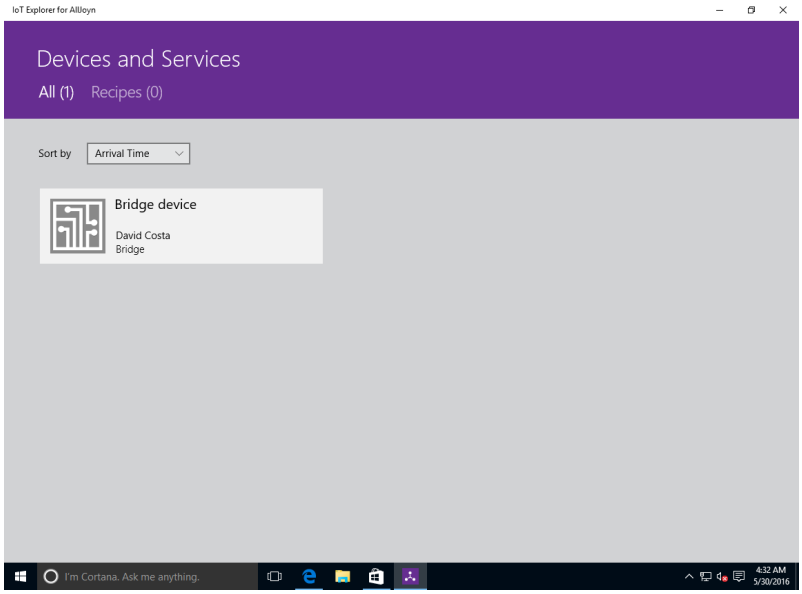


Figure 39: IoT Explorer for AllJoyn: Discovery

7.2.2 *Objects*

By clicking on any of the service tiles, the application shows more detailed information about that service. After selecting the *Bridge device*, the application shows the AllJoyn objects exposed by the bridge (Figure 40). Each tile shown on this page represents a single bus object that can be interacted with using IoT Explorer for AllJoyn. In this test, there are all the CoAP resources provided by the Wireless Sensor Network, after that every CoAP device registered on the bridge.

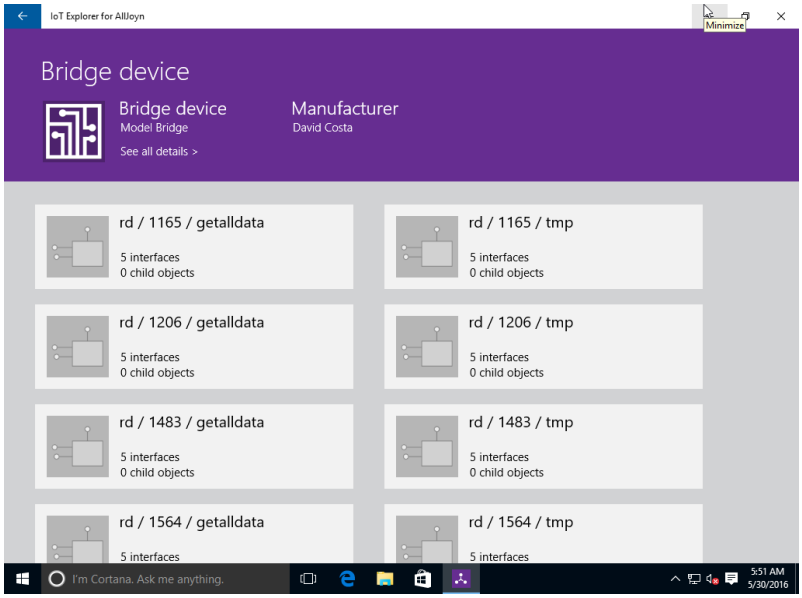


Figure 40: IoT Explorer for AllJoyn: Objects

7.2.3 *Interfaces*

We can get a more detailed information about a bus object by clicking on the corresponding tile. At the top of the page, we see the AllJoyn service name and bus object name, and at the bottom of the page we see tiles corresponding to each of the interfaces that this bus object exposes. Each interface can expose a number of properties, methods, and signals for this bus object. We can see in Figure 41 the *com.bridge.Coap* interface, among other ones.

7.2.4 *Methods, signals, properties*

Clicking on any of the interface tiles lets you see more information about the properties, methods, and signals that they expose. Figure 42 display the methods, the signals and the properties provided by the *com.bridge.Coap* interface. This interface exposes two properties allowing to read the Resource Type and the Interface Description of the resource, the REST methods *get*, *post* and *delete*, and the members related to the observing service.

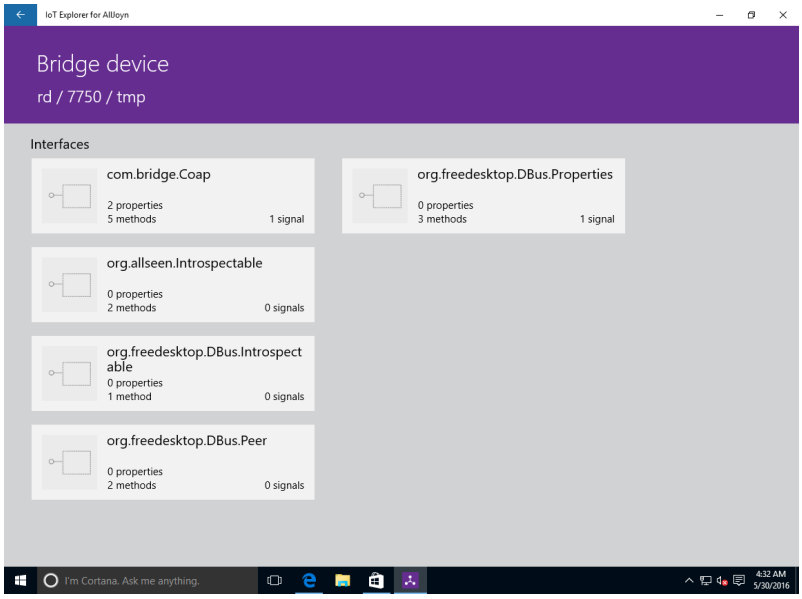


Figure 41: IoT Explorer for AllJoyn: Interfaces

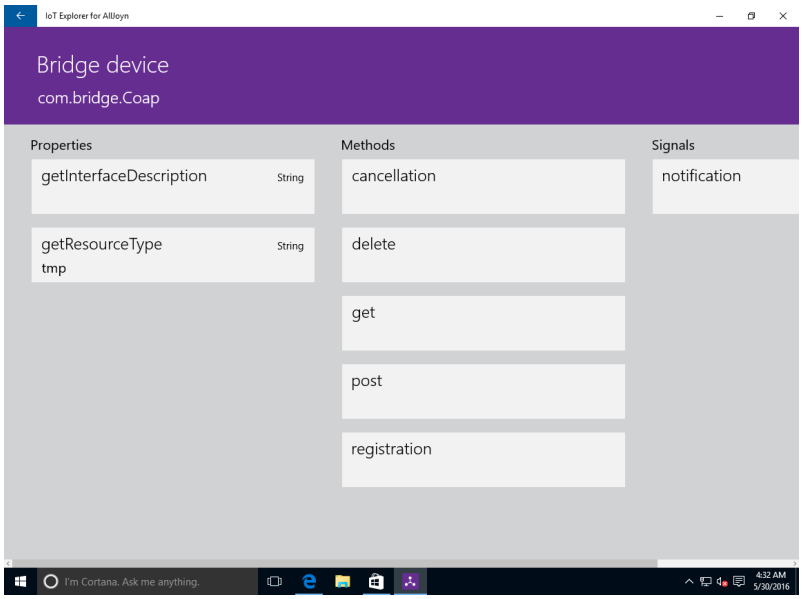
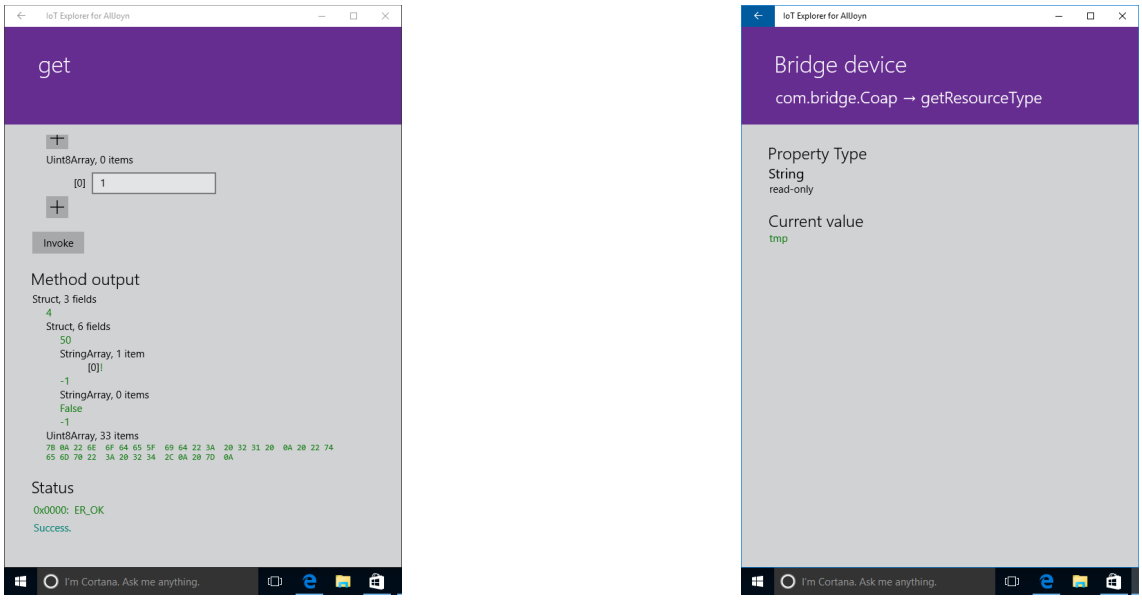


Figure 42: IoT Explorer for AllJoyn: Methods

7.2.5 *Method call*

When you click on a tile corresponding to a method, you are taken to a page with more details about that method. The page will indicate what input parameters the method takes, including their types and names (if available). It also gives information about the method's return parameters. After entering the input parameters in the provided fields, you can invoke the method with the *Invoke* button. Instead, when you click on a property tile, you will be taken to a more detailed view of that property. You will be able to see the type, the current value of the property, and if the property is writable, you will be able to set a new value.

For example, Figure 43 shows what happen during a method call and a property call. In particular, the response received after the *get* method has been called (Figure 43a) and the value of the *Resource Type* property (Figure 43b).



(a) Get Method

(b) Resource Type Property

Figure 43: IoT Explorer for AllJoyn: Method Call

7.3 OBSERVING SERVICE

An important feature of the Constrained Application Protocol is the observing service, which allows client nodes to retrieve a representation of a resource and keep this representation updated by the server as long as the client is interested. The observing service is implemented in AllJoyn using signals.

The devices used in the test case described above provide observable resources, so that an AllJoyn client application can receive the resource representation over time. The CoAP device contains one temperature sensor and one light sensor, and provides both them using a CoAP resource, which sends notifications every minute. The consumer application registered its interest into receiving the resource updates, and it registers a

signal handler in order to manage the received notifications. Figure 44 illustrates how the light value changes, sensed by the CoAP device's sensor, at the end of a 30 minutes test.

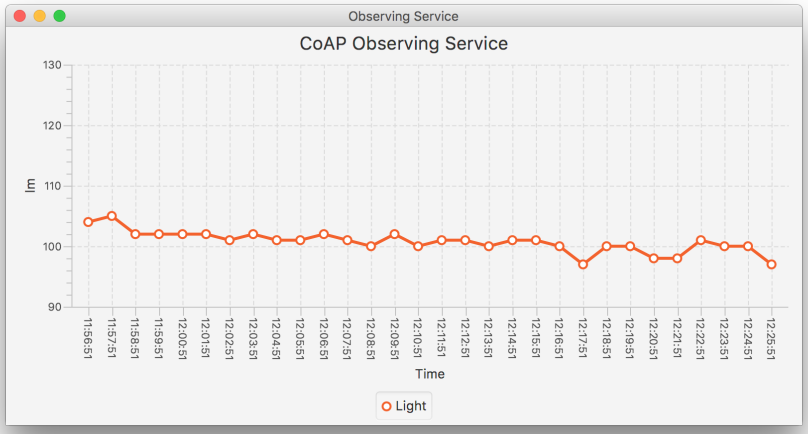


Figure 44: Observing Service in Test Case

CONCLUSIONS

In this work an AllJoyn to CoAP bridge application has been presented. The work starts off by surveying the AllJoyn framework architecture and the state of the art of the building automation. It includes the observed limitations of these advanced software frameworks, with focus on the interoperability with the Constrained Application Protocol. The survey then presents the bridging solution proposed by Microsoft and the reason for the development of our own bridge.

The proposed solution is designed to enable CoAP resources in low-power devices to be reached by AllJoyn applications, without the need of further configurations. Resource-constrained nodes based on CoAP can easily register their resources on the bridge, which provides to translate and advertise these resources, so that AllJoyn consumer application are enabled to discover and interact with them.

Experimental results carried out by means of both a simulated testbed and a real testbed demonstrated that the proposed solution succeeds in extending the AllJoyn framework.

The developed solution enables AllJoyn to communicate with the CoAP protocol in a low level of abstraction, setting itself in the CoAP message exchange level. Related works can easily exploit the proposed bridge in order to implement applications based on it, addressed to specific use cases.

REFERENCES

[1] Gartner, "Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015," 2015. [Online]. Available: http://www.gartner.com/newsroom/id/3165317.

[2] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252, RFC Editor, June 2014.

[3] A. Alliance, "The innovative companies that support AllJoyn," 2016. [Online]. Available: https://allseenalliance.org/alliance/members.

[4] A. Alliance, "AllJoyn Framework," 2016. [Online]. Available: https://allseenalliance.org/framework.

[5] H. Pennington, A. Carlsson, A. Larsson, S. Herzberg, S. McVittie, and D. Zeuthen, "D-Bus Specification," tech. rep., February 2015.

[6] A. Alliance, "Documentation," 2016. [Online]. Available: https://allseenalliance.org/framework/documentation.

[7] C. Partridge and R. Hinden, "Version 2 of the Reliable Data Protocol (RDP)," RFC 1151, RFC Editor, April 1990.

[8] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, RFC Editor, August 2008.

[9] D. McGrew and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)," RFC 6655, RFC Editor, July 2012.

[10] Microsoft, "AllJoyn Device System Bridge," 2016. [Online]. Available: http://ms-iot.github.io/content/en-US/win10/AllJoynDSB.htm.

[11] Microsoft, "AllJoyn Device System Bridge Template," 2016. [Online]. Available: https://visualstudiogallery.msdn.microsoft.com/aea0b437-ef07-42e3-bd88-8c7f906d5da8.

[12] Microsoft, "Mapping Bridge Interface Objects to Alljoyn," 2016. [Online]. Available: http://ms-iot.github.io/content/en-US/win10/AlljoynDsbApiGuide.htm.

[13] Microsoft, "ZigBee Adapter," 2016. [Online]. Available: http://ms-iot.github.io/content/en-US/win10/samples/ZigBeeAdapterTutorial.htm.

[14] Microsoft, "BACnet Sample," 2016. [Online]. Available: http://ms-iot.github.io/content/en-US/win10/samples/BACnetAdapterTutorial.htm.

[15] Z. Shelby, "Constrained RESTful Environments (CoRE) Link Format," RFC 6690, RFC Editor, August 2012.

[16] K. Hartke, "Observing Resources in the Constrained Application Protocol (CoAP)," RFC 7641, RFC Editor, September 2015.

[17] A. Alliance, "AllJoyn Code Generator," 2015. [Online]. Available: `https://wiki.allseenalliance.org/devtools/code_generator`.

[18] Microsoft, "IoT Explorer for AllJoyn," 2015. [Online]. Available: `https://www.microsoft.com/it-it/store/apps/iot-explorer-for-alljoyn/9nblggh6gpxl`.