Introduction to AllJoyn™

HT80-BA013-1 Rev. C

February 8, 2013

Submit technical questions at:

http://www.alljoyn.org/forums

Qualcomm Innovation Center, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Figures

Tables

1 Preface

1.1 Purpose

This document provides a high-level introduction to AllJoyn™. It describes the overall goals and benefits of the system, and how AllJoyn is expected to enhance the experience of networking in a mobile environment. Additionally, this document covers the basic abstractions of the system and describes, at a high level, how the various components of an AllJoyn-enabled system work together to provide an environment for proximity-based peer-to-peer mobile computing.

1.2 Scope

This document is written for anyone interested in understanding how an AllJoyn-enabled system can enhance networked or distributed mobile applications. We assume no particular expertise in mobile communications, so this document will be appropriate for anyone interested in peer-to-peer networks, including application developers, system software developers, network specialists, device manufacturers, and systems managers.

1.3 Revision history

The table below provides the revision history for this document.

Table 1: Revision History

Version	Date	Description
A	June 22, 2011	Initial release
B	November 22, 2011	Formatting changes only
C	February 8, 2013	Updated with information about additional OS and language support. Updated documentation list.

2 Overview

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

AllJoyn is “platform-neutral,” meaning it was designed to be as independent as possible of the specifics of the operating system, hardware, and software of the device on which it is running. In fact, AllJoyn was developed to run on Microsoft Windows, Linux, Android, iOS, OS X, OpenWRT, and the Unity plug-in for internet browsers.

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

AllJoyn is being developed as an open source project licensed under the Apache Version 2.0 license. The AllJoyn codebase is available at *http://www.alljoyn.org*.

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

The majority of handsets today have Bluetooth integrated, so if two people are in the same room and have Bluetooth turned on, AllJoyn can enable direct communication between the phones and allow applications to interact (if the AllJoyn security system allows it). If, for example, these two users walk into a home or office that has a Wi-Fi hotspot, their devices can connect to the underlying access point and transparently take advantage of the additional network capacity. Additionally, their devices can locate other devices in the proximity (defined by the Wi-Fi coverage footprint), can discover additional services on the other devices, and use those services, if desired.

Enabling real-time multi-player gaming is another example of how AllJoyn might be used. *Figure 1* shows how a multi-user game can be accomplished using different device classes and different underlying network technologies. The details of the infrastructure management are all handled by AllJoyn, allowing the game author to focus on the design and implementation of the game, rather than dealing with the complexities of the peer-to-peer networking.

Figure 1: AllJoyn heterogeneous networking

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

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

The possibilities are endless.

3 Benefits of AllJoyn

As mentioned, AllJoyn is a platform-neutral system that is designed to simplify proximity networking across heterogeneous distributed mobile systems. Heterogeneous in this case means not only different devices, but different kinds of devices (e.g., PCs, handsets, tablets, consumer electronics devices) running on different operating systems, using different communication technologies.

3.1 Open source

AllJoyn is being developed as an open source project licensed under the Apache Version 2.0 license. This means that all of the AllJoyn codebase is available for inspection, and developers are encouraged to contribute additions and enhancements. If AllJoyn is missing a feature, you can contribute. If you run into a snag using AllJoyn, or have a technical question, other participants in the open source community are ready and willing to provide help and guidance. The AllJoyn codebase is available at *http://www.alljoyn.org*.

3.2 Operating system independence

AllJoyn provides an abstraction layer allowing AllJoyn and its applications to run on multiple OS platforms. As of this writing, AllJoyn supports most standard Linux distributions including Ubuntu, and runs on Android 2.2 and later smartphones and tablets. AllJoyn also runs and is tested and validated on commonly available versions of the Microsoft Windows operating system including Windows XP, Windows 7, Windows RT, and Windows 8. Additionally, AllJoyn runs on Apple operating systems iOS and OS X, on embedded operating systems such as OpenWRT, and in an internet browser it works with the Unity plug-in.

3.3 Language independence

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

3.4 Physical network and protocol independence

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

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

3.5 Dynamic configuration

Often, as a mobile device makes its way through the various locations it encounters during its lifetime, associations with networks may come and go. This means that IP (Internet Protocol) addresses may change, network interfaces may become unusable, and services may be transitory.

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

There are situations where network topologies are critical to the performance of distributed applications. Bluetooth networks perform much better when configured as piconets than when configured as scatternets. AllJoyn manages this internally without any need for developers to understand the individual traits specific to each networking technology.

3.6 Service advertisement and discovery

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

3.7 Security

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

3.8 Object model and remote method invocation

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

3.9 Software componentry

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

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

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

4 Conceptual Overview

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

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

4.1 Remote Method Invocation

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

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

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

4.2 AllJoyn bus

The most basic abstraction of the AllJoyn system is the AllJoyn bus. It provides a fast, lightweight way to move marshaled messages around the distributed system. One can view the AllJoyn bus as a kind of “freeway” over which those messages flow. *Figure 2* shows what an instance of an AllJoyn bus on a single device might look like, conceptually. The bus itself is shown as the thick horizontal dark line. The vertical lines can be thought of as “exits” and are the sources and/or destinations of messages that are flowing over the bus.

The connections to the bus shown in *Figure 2* are depicted as hexagons (an arbitrarily chosen shape used in this document). Just as the exits on a freeway are typically assigned numbers, each connection is assigned a unique connection name. A simplified form of the connection name is used here for clarity.

In many cases, the connections to the bus can be thought of as co-resident with processes. Therefore, in the example of *Figure 2*, the unique connection name `:1.1` may be assigned to a connection in a process running some instance of an application, and the unique connection name `:1.4` may be assigned to a connection in a process running an instance of some other application. The goal of the AllJoyn bus is to allow the two applications to communicate without having to deal with the details of the underlying mechanisms. One of the connections can be thought of as the client stub, and the other side can fulfill the duties of the service stub.

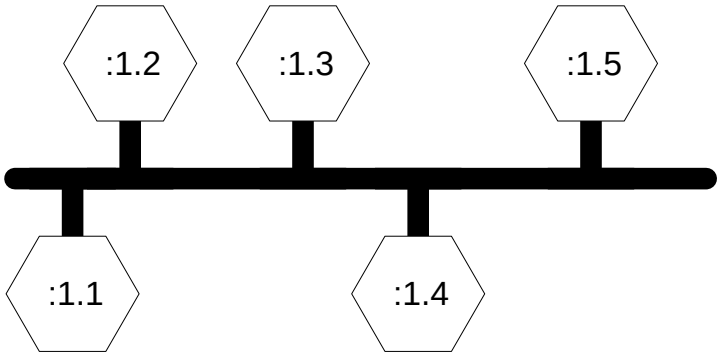

Figure 2: A prototypical AllJoyn bus

Figure 2 shows an instance of an AllJoyn bus and illustrates how a software bus can provide interprocess communication between components attached to the bus. The AllJoyn bus is typically extended across devices as shown in *Figure 3*. A communication link between the segment of the logical bus residing on the Smartphone and the components residing on the Linux host is formed when required by the components.

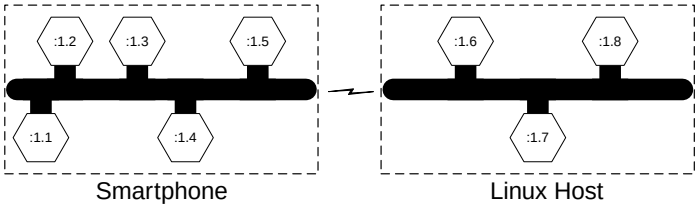

Figure 3: AllJoyn handles device-to-device communication

The management of this communication link is handled by the AllJoyn system and may be formed using a number of underlying technologies such as Wi-Fi and Bluetooth. There may be multiple devices involved in hosting the AllJoyn bus, but this is transparent to the

users of the distributed bus. To a component on the bus, a distributed AllJoyn system looks like a bus that is local to the device.

Figure 4 shows how the distributed bus may appear to a user of the bus. A component (for example, the Smartphone connection labeled `:1.1`) can make a procedure call to the component labeled `:1.7` on the Linux host without having to worry about the location of that component.

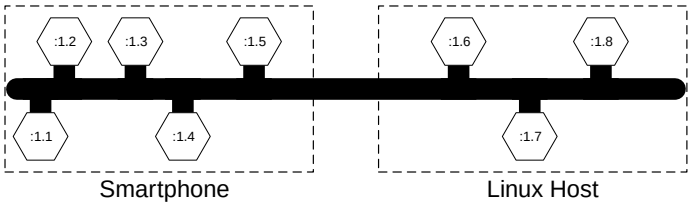

Figure 4: A distributed AllJoyn bus appears as a local bus

4.3 Bus daemons

Figure 3 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 daemon.

The term daemon is commonly used in Unix-derived systems to describe programs that run to provide some needed functionality to the computer system. In Windows systems, the term service is more typically used, however one still refers to the AllJoyn daemon on a Windows system.

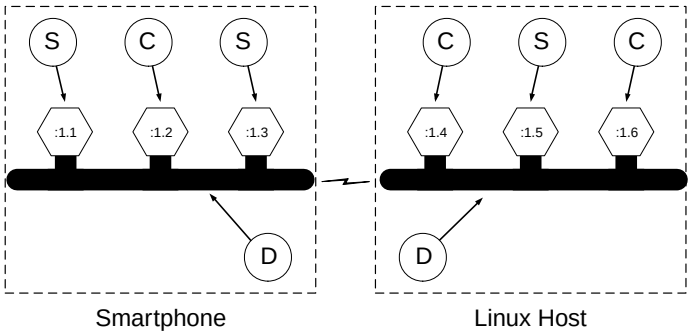

Figure 5: Relating bubble diagrams to the bus

In order to visualize the AllJoyn daemons, it is useful to create a bubble diagram. Consider two AllJoyn bus segments, one residing on a Smartphone and one on a Linux Host, as shown in *Figure 5*. The connections to the bus are labeled as clients (C) and services (S) using the sense of clients and services in the RMI model. The daemon programs that implement the core of the distributed bus are labeled (D). The components of *Figure 5* are typically translated into the illustration shown in *Figure 6*.

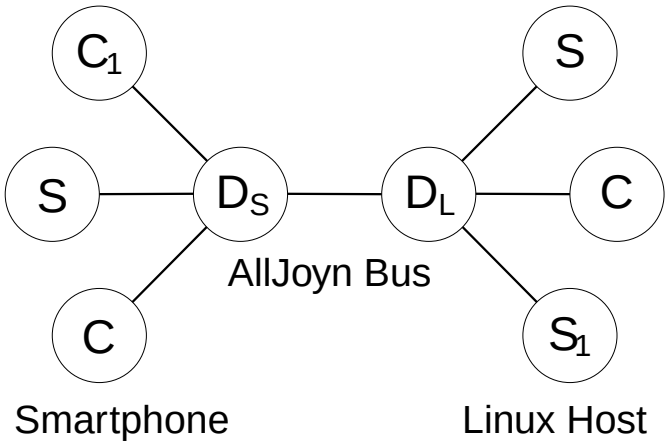

Figure 6: An AllJoyn bubble diagram

The bubbles can be viewed as computer processes running on a distributed system. The two client (C) and the service (S) processes on the left are running on the Smartphone. These three processes communicate with an AllJoyn daemon running on the Smartphone which implements the local segment of the distributed AllJoyn bus. On the right side, there is a daemon which implements the local segment of the AllJoyn bus on the Linux Host. These two daemons coordinate the message flow across the logical bus, which appears as a single entity to the connections, as shown in *Figure 4*. Similar to the configuration on the Smartphone, there are two service components and a client component on the Linux host.

In this configuration, client component C_1 can make remote method calls to service component S_1 as if it were a local object. Parameters are marshaled at the source and routed off of the local bus segment by the daemon residing on the Smartphone. The marshaled parameters are sent over the network link (transparently from the perspective of the client) to the daemon on the Linux host. The daemon running on the Linux host determines that the destination is S_1 and arranges to have the parameters unmarshaled and the remote method invoked on the service. If return values are expected, the process is reversed to communicate the return values back to the client.

Since the daemons are running in a background process and the clients and services are running in separate processes, there must be a “representative” of the daemon in each of those separate processes. AllJoyn calls these representatives *bus attachments*.

4.4 Bus attachments

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

An analogy is often drawn between hardware and software when discussing software components. One can think of a local segment of a distributed AllJoyn bus in much the

same way as one thinks of the hardware backplane bus in a desktop computer. The hardware bus itself moves electronic messages and has attachment points called connectors into which one plugs cards. The analogous function of the connector in AllJoyn is the bus attachment.

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

4.5 Bus methods bus properties and bus signals

AllJoyn is fundamentally an object-oriented system. In object-oriented systems, one speaks of invoking methods on objects (thus the term Remote Method Invocation when speaking of distributed systems). Objects in the object-oriented programming sense have members. Classically, these are object methods or properties, which are known as BusMethods and BusProperties in AllJoyn. AllJoyn also has the concept of a BusSignal, which is an asynchronous notification of some event or state change in an object.

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

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

4.6 Bus interfaces

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

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

Interface names typically take the form of a reversed domain name. For example, there are a number of standard interfaces that AllJoyn implements. One of the AllJoyn standard interfaces is the `org.alljoyn.Bus` interface which daemons implement and which provides some of the basic functionality for bus attachments.

It is worth noting that the interface name is simply a string in a relatively free-form namespace and that other namespaces may have a similar look. The interface name string serves a specific function that should not be confused with other similar strings, in particular bus names. For example, `org.alljoyn.sample.chat` may be a bus name which is the constant, unchanging name that a client will search for. It may also be the case that `org.alljoyn.sample.chat` is the interface name that defines the methods, signals and properties available in a bus object associated with a bus attachment of the specified bus name. The existence of an interface with the given interface name is implied by the existence of the bus name; however, they are two completely different things that can sometimes look exactly the same.

4.7 Bus objects and object paths

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

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

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

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

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

4.8 Proxy bus object

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

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

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

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

4.9 Bus names

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

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

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

To use a well-known name, an application (by way of a bus attachment) must make a request of the bus daemon to use that name. If the well-known name is not already in use

by another application, exclusive use of the well-known name is granted. This is how, at any time, well-known names are guaranteed to represent unique addresses on the bus.

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

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

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

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

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

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

4.10 Advertisements and discovery

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

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

The answer is through the AllJoyn service advertisement and discovery facility. When a service is started on a local device, it reserves a given well-known name and then advertises its existence to other devices in its proximity. AllJoyn provides an abstraction layer that makes it possible for a service to do an advertise operation that may be communicated

transparently via underlying technologies, such as Wi-Fi, Bluetooth, or Wi-Fi Direct. Neither the client nor the service require any knowledge of how these advertisements are managed by the underlying technology.

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

Client applications may declare their interest in receiving advertisements by initiating a discovery operation. For example, it may ask to discover instances of the contacts service as specified by the prefix `org.alljoyn.sample.contacts`. In this case, both devices would make that request.

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

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

4.11 Sessions

The concepts of bus names, object paths, and interface names have been previously discussed. Recall that when an entity connects to an AllJoyn bus, it is assigned a unique name. Connections (bus attachments) may request that they be granted a well-known name. The well-known name is used by clients to locate or discover services on the bus. For example, a service may connect to an AllJoyn bus and be assigned the unique name `:1.1` by the bus. If a service wants other entities on the bus to be able to find it, the service must request a well-known name from the bus, for example, `com.companyA.ProductA` (remember that a unique instance qualifier is usually appended).

This name implies at least one bus object that implements some well-known interface for it to be meaningful. Usually, the bus object is identified within the connection instance by a path with the same components as the well-known name (this is not a requirement, it is only a convention). In the example, the path to the bus object corresponding to the bus name `com.companyA.ProductA` might be `/com/companyA/ProductA`.

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

4.11.1 Postal address analogy

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

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

Unique name::1.1
```

This tells us that the well-known name advertised as `org.alljoyn.sample.chat` corresponds to a bus attachment that has been assigned the unique name `:1.1`. One can think of this in the same way as a business has a name and a postal address.

To continue the analogy, a common situation arises when a business is located in a building along with other businesses. In such a situation, one might find a business address further qualified by a suite number. Since AllJoyn bus attachments are capable of providing more than one service, there must also be a way to identify more than one destination on a particular attachment. A “contact port number” corresponds to the suite number destination in the postal address analogy.

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

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

4.11.2 The AllJoyn session

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

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

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

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

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

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

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

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

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

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

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

Notice that there are two instances of the session options. When communication establishment begins, these may be viewed as supported session options provided by the service and requested session options provided by the client. Part of the session establishment procedure consists of negotiating an actual final set of options to be used in the session. Once a session has been formed, the half-associations of the client and service side describe a unique AllJoyn communication path:

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

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

Because issues regarding end-to-end flow control of the underlying connection must be balanced with topological concerns in some technologies, the actual connection between the two communicating endpoints (the “from” client and the “to” service) may or may not result in a separate communication channel being formed. In some cases it is better to flow messages over an ad-hoc topology (Bluetooth piconets) and in some cases it may be better to flow messages directly over a new connection (TCP-IP). This is another of the situations that may require deep understanding of the underlying technology to resolve, and which

AllJoyn happily accomplishes for you. A user need only be aware that messages are routed correctly over a transport mechanism that meets the abstract needs of the application.

4.12 Bringing it all together

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

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

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

Figure 7 shows a hypothetical arrangement of how all of these pieces are related. At the center is the dark line representing the AllJoyn bus. The bus has “exits” which are the BusAttachments assigned the unique names `:1.1` and `:1.4`. In the figure, the BusAttachment with the unique name of `:1.1` has requested to be known as `org.alljoyn.samples.chat.1` and has been assigned the corresponding well-known bus name. The “1” has been added to ensure that the bus name is unique.

There are a number of things implied by taking on that bus name. First, there is a tree structure of bus objects that resides at different paths. In this hypothetical example, there are two bus objects. One is at the path `/org/alljoyn/samples/chat/chat` and which presumably implements an interface suitable for chatting. The other bus object lives at the path `/org/alljoyn/samples/chat/contacts` and implements an interface named `org.alljoyn.samples.chat.contacts`. Since the given bus object implements the interface, it must provide implementations of the corresponding bus methods, bus signals, and bus properties.

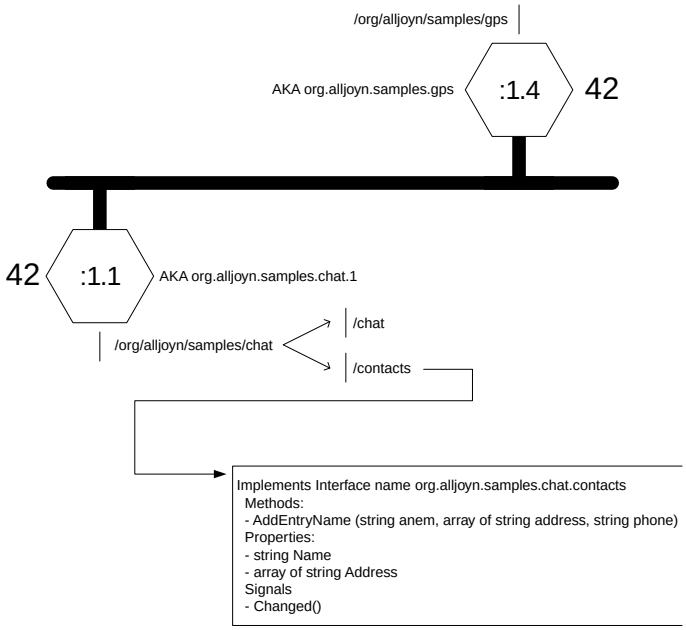

Figure 7: Overview of a hypothetical AllJoyn bus instance

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

After requesting and being granted the well-known bus name, a service will typically advertise the name to allow clients to discover its service. *Figure 8* shows a service making an advertise request to its local daemon. The daemon, based on input from the service, decides what network medium-specific mechanism it should use to advertise the service and begins doing so.

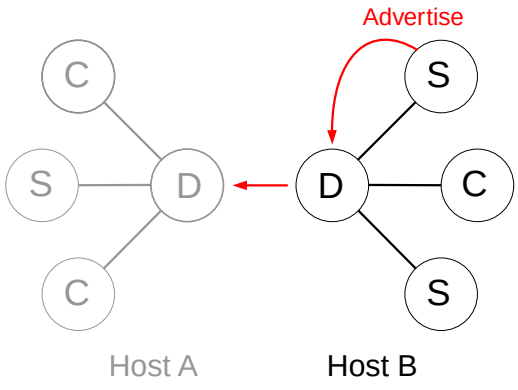

Figure 8: Service performs an Advertise

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

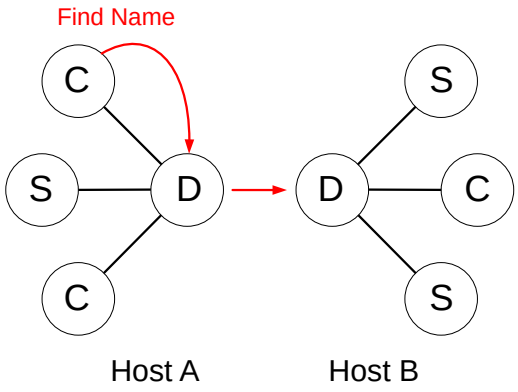

Figure 9: Client requests to Find Name

Once the devices move into proximity, they begin hearing each other's advertisements and discovery requests over whichever media are enabled. *Figure 10* shows how the daemon hosting the service hears the discovery requests and responds.

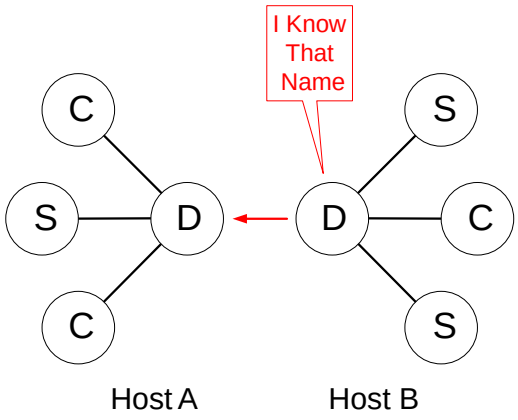

Figure 10: Daemon reports Found Name

Finally, *Figure 11* shows the client receiving an indication that there is a new daemon in the area that is hosting the desired service.

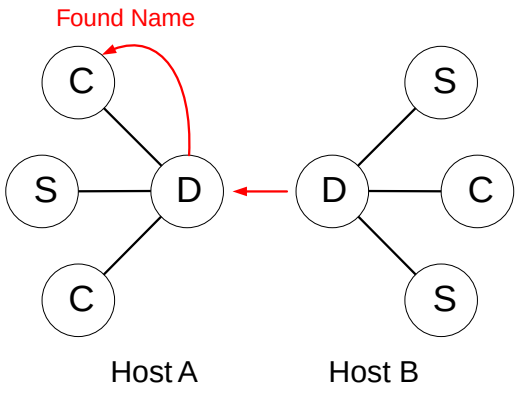

Figure 11: Client discovers service

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

Before remote methods can be called, a communication session must be formed to effectively join the separate bus segments. Advertisement and discovery are different from session establishment. One can receive an advertisement and take no action. It is only when an advertisement is received, and a client decides to take action to join a communication session, that the busses are logically joined into one. To accomplish this, a service must create a communication session endpoint and advertise its existence; and a client must receive that advertisement and request to join the implied session.

The service must define a half-association before it advertises its service. Abstractly this will look something like the following:

```
{reliable IP messages, org.alljoyn.samples.chat.1, 42}
```

This indicates that it will talk to clients over a reliable message-based transport, has taken the well-known bus name indicated, and expects to be contacted at session port 42. This is the situation seen in *Figure 7*.

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

```
{reliable IP messages, org.alljoyn.samples.chat.1, :2.1, 1025}
```

The new communication session will use a reliable messaging protocol implemented using the IP protocol stack which will exist between the bus attachment named `org.alljoyn.samples.chat.1` (the service) and the bus attachment named `:2.1` (the

client). The session ID used to describe the session is assigned by the system and is 1025 in this case.

As a result of establishing the end-to-end communication session, the AllJoyn system takes whatever actions are appropriate to create the virtual software bus shown in *Figure 4*. Note that this is a virtual picture, and what may have actually happened is that a Wi-Fi Direct peer-to-peer connection was formed to host a TCP connection, a Wireless access point was used to host a UDP connection, or a Bluetooth piconet was formed to host an L2CAP connection, depending on the provided session options. Neither the client nor the service is aware that this possibly very difficult job was completed for them.

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

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

5 High-Level System Architecture

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

5.1 Clients, services, and peers

Figure 12 shows the architecture of the system from a user (not daemon) perspective. At the highest level are the language bindings. The AllJoyn system is written in C++, so for users of this language, no bindings are required. However, for users of other languages, such as Java or JavaScript, a relatively thin translation layer called a language binding is provided. In some cases, the binding may be extended to offer system-specific support. For example, a generic Java binding will allow the AllJoyn system to be used from a generic Java system that may be running under Windows or Linux; however, an Android system binding may also be provided which more closely integrates the AllJoyn system into Android-specific constructs such as a service component in the Android application framework.

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

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

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

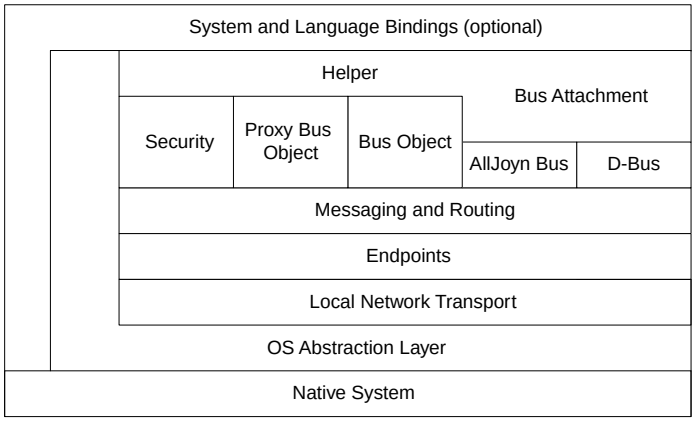

Figure 12: The basic client, service, or peer architecture

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

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

5.2 Daemons

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

At the lowest level of *Figure 13*, resides the native system. We use the same OS abstraction layer as we do in the client architecture to provide common abstractions for daemons running on Linux, Windows, and Android. Running on the OS abstraction layer, we have the various low-level networking components of the daemon. Recall that clients, services, and peers only use a local interprocess communication mechanism to talk to a daemon, so it is the daemon that must deal with the various available transport mechanisms on a given platform. Note the “Local” transport in *Figure 13* which is the sole connection to the AllJoyn clients, services, and peers running on a particular host.

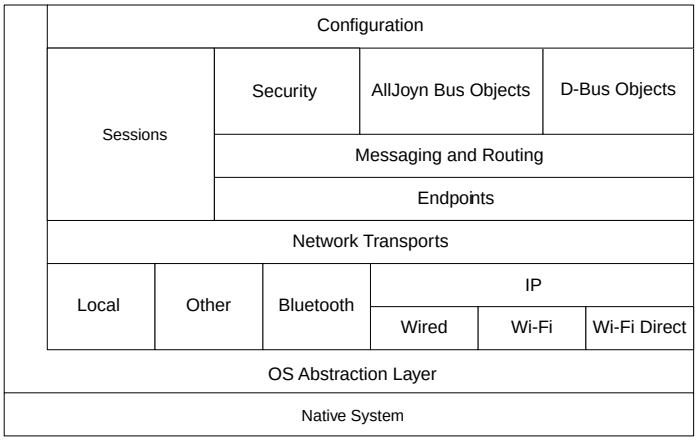

Figure 13: The basic daemon architecture

The Bluetooth transport handles the complexities of creating and managing piconets in the Bluetooth system. Additionally, the Bluetooth transport provides service advertisement and discovery functions appropriate to Bluetooth, as well as providing reliable communications.

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

The various technology-specific transport implementations are collected into a Network Transports abstraction. The Sessions module handles the establishment and maintenance of communication connections to make a collection of daemons and their associated clients, services, and peers appear as a unified software bus.

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

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

The overall operation of certain aspects of daemon operation are controlled by a configuration subsystem. This allows a system administrator to specify certain permissions for the system and provides the ability to arrange for on-demand creation of services. Additionally, resource consumption may be limited by configuration of the daemon, allowing

a system administrator to, for example, limit the number of TCP connections active at any given time. There are options which allow system administrators to mitigate the effects of certain denial-of-service attacks, by limiting the number of connections which are currently authenticating, for example.

6 Summary

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

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

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

7 Learn More

To learn more about how to integrate AllJoyn in your development efforts, access the following documentation on the AllJoyn web site (*https://www.alljoyn.org/docs-and-downloads*).

Introductory Guides: The documents in this category describe AllJoyn technologies and concepts.

- Introduction to AllJoyn (this document)

Development Guides: Development guides provide solutions to specific programming problems and guidelines to setting up the build environment. They also include code snippets with explanations.

- AllJoyn Android Environment Setup Guide
- Guide to AllJoyn Development Using the Java SDK
- AllJoyn Android C++ Sample Programs Walkthrough
- AllJoyn C# Chat Sample Walkthrough
- AllJoyn WinJS Chat Sample Walkthrough
- AllJoyn Unity Setup Quick Start Guide
- AllJoyn Troubleshooting Guide
- AllJoyn Programming Guide for the Objective-C Language
- Configuring the Build Environment (Microsoft Windows XP and Windows 7)
- Configuring the Build Environment (Linux Platform)
- Configuring the Build Environment (Windows 8)
- Configuring the Build Environment (iOS and OS X)

API References: API References provide details for working with the AllJoyn source code and writing applications in each supported programming language.

- Java API Reference
- C++ API Reference
- Windows RT API Reference
- JavaScript API Reference
- C# Unity API Reference

Downloads: A software developers kit (SDK) and sample code are available to users to help build, modify, test, and execute specific tasks.

- AllJoyn SDK (including the AllJoyn daemon) with sample code
- AllJoyn C++ Code Generator Source