OTA Firmware Update – High Level Design

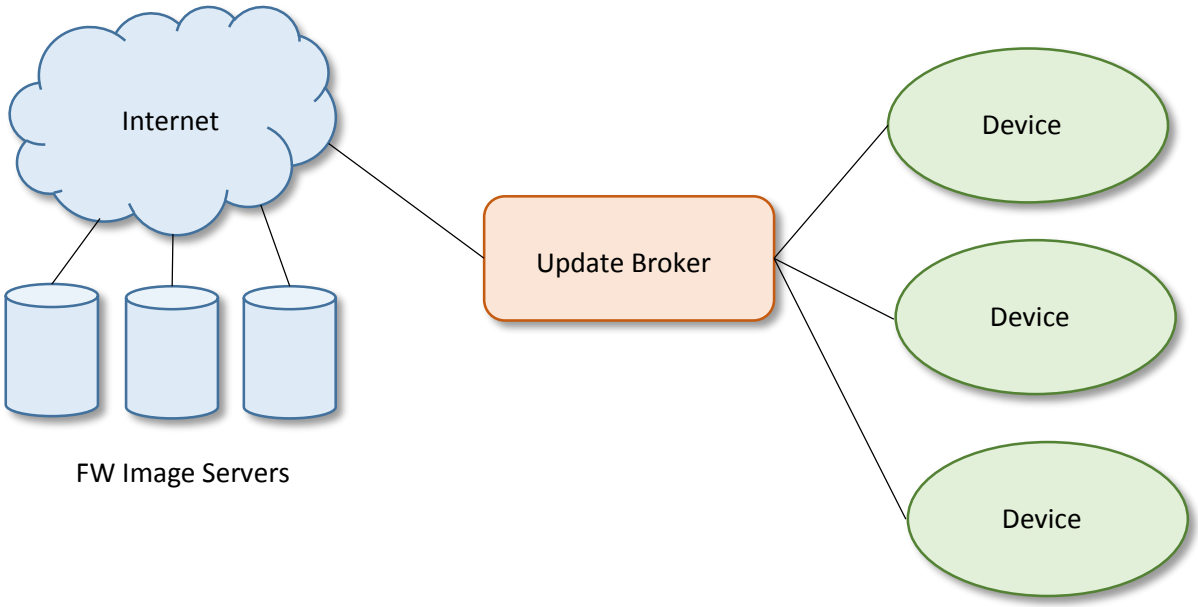
A system to enable over the air firmware updates using AllJoyn comprises of 3 major components and 2 primary communication relationships. The major components are as follows:

- Firmware image providers hosts on servers accessible from the Internet.
- An update broker that retrieves updates from the firmware image providers.
- The devices to be updated.

The major communication relationships are:

- Firmware image provider to update broker.
- Update broker to device being updated.

The diagram below provides a graphical representation.



As can be seen, the devices being updated do not directly communicate with the firmware image providers. This actually allows for a uniform firmware update design that would work for any transport supported by AllJoyn. This could even include providing firmware updates for devices that do not have any kind of connection to the Internet but are still part of the local AllJoyn network. This could potentially include devices that only communicate over Bluetooth, ZigBee, or ZWave, for example.

The first part of this document will discuss the communications while the second part will discuss the major components.

Components

Device

This is the device with the firmware to be updated. It must host a session port for the update broker to join when broker has found a suitable firmware update. The device must announce itself using About.

It will need to include a list of SHA256 hash for all updatable firmware images. Most devices will only have 1 updatable firmware image, but there may be some devices that have multiple sub-systems where each sub-system has its own firmware image.

Update broker

The update broker sits between the device and firmware image provider. It's job is to actively look for firmware updates, locally cache the firmware updates, and distribute the cached image to all devices that image applies to.

Consumer update broker

A consumer update broker is a simple autonomous program that runs on an "always-on" device with sufficient storage to cache firmware images. It just runs silently in the background.

Commercial update broker

A commercial update broker is more sophisticated than its consumer counterpart. While it still automatically looks for firmware updates and caches them locally, it does not automatically update devices on the AllJoyn network. Instead, it will inform the IT department of an update. This will allow for the IT department to vet the update and to schedule when the update gets applied.

Additional capabilities could be to limit which devices a broker will update or even from which firmware image provider updates will be retrieved. These would allow IT to deploy the update in phases or to secondary brokers for load balancing, enable local development and testing of the devices themselves or even provide additional security by limiting where image come from.

Lastly, a commercial update broker should allow the IT department to override which firmware image ID is considered the latest to enable the ability for IT to rollback an update in the event a problem is discovered after deployment.

Firmware image provider

Generally speaking this would a website known to update brokers to host firmware images and provide a mechanism to query or otherwise discover the existence of updated firmware images.

Communications

Update broker to device

Communications between the device and the update broker shall exclusively use the AllJoyn protocol defined here. There are 2 parts to communications between update brokers and devices: image discovery and image data transfer.

Image discovery

There are 2 phases to image discovery. The first is for the update broker to learn about all the devices that potentially need to be updated and their current firmware images. The About discovery mechanism already contains the software version information the update broker needs. All that needs to be added to About is an indication that the device can use AllJoyn for firmware update so that the update broker can monitor for software updates for that device.

Another important aspect of firmware image discovery is image identification. The OTA firmware update over AllJoyn process will use the SHA256 hash of the firmware image as the sole identifier for all

firmware images. A SHA256 hash is sufficiently unique that it can uniquely identify a single firmware image across all HW designs, all manufacturers, all OEM personalizations, along with all versions for a given HW and SW design.

In order for the update broker to query firmware update providers on the Internet, it first needs to enumerate all the updatable devices on the local AllJoyn network and the current version of firmware/software that each updatable device is currently running.

Since most devices that will make use of the AllJoyn firmware update mechanism will already be sending out About announcements, the firmware update process will take advantage of that. Any device that wants to be updated via AllJoyn will need to add an entry to the About data dictionary:

- Key: AllJoynFirmwareUpdate
- Value: A struct containing a session port number and a list of SHA256 values. The signature would be "(qaay)".

The session port number is for the updatable device to host a session from the update broker that will be used to notify the device of an available update and for transfer of the firmware image. Each "ay" byte array in the list of SHA256 values contains a full SHA256 value, therefore, it must be exactly 32 bytes in length. Any byte arrays that are either shorter or longer must be ignored.

The purpose of providing an array of SHA256 values is to support devices that have multiple independent firmware images that need to be updated. While most devices will only have 1 firmware image for the whole device, some device designs may require multiple different images. On possibility, for example, would be a device with separate processing units that handle different aspects of the device but do not have direct access to the AllJoyn network themselves.

When an update broker determines that it has a firmware update for a device it will create a session with that device using that device's firmware update session port. Once the session is established, the update broker will send a notification signal to that device over the newly established session.

If the update broker fails to establish a session with a device for the purpose of updating, it shall retry 1 hour later. A second retry would happen 6 hours after that. A third retry 12 hours after that. And finally, a fourth retry 24 hours after that. After a 4th failed attempt to establish a session for updating, the update broker will remove that device from its list of devices that it serves. If the device updates its About data with an new SHA256 value for its firmware image ID before all retries have been exhausted then the update broker will cease further retries and use the new SHA256 image ID for check for updates.

Devices should only accept a single session joiner on the firmware update session port.

Image data transfer

The transfer of the actual image data from the update broker to the device will use a generic bulk data transfer protocol defined (elsewhere in this document?). The device will take the role of client and the update broker will take the role of server.

The device will make a method call requesting an update to its current firmware image. Upon receipt of that request, the update broker will either create a new bus object that implements the bulk data transfer interface for that firmware image or it will up the reference count to a pre-existing bus object

for that firmware image. Once the bus object is available, the update broker will respond with an object path to that bus object (along with the SHA256 value of the new image?). The update broker will keep that firmware image in its cache and available until all users of the associated bus object are done.

The client will issue get requests with the starting offset and desired payload size. The server will respond with a payload that is no larger than the requested size. The server should try its best to fill payload with the requested size to minimize traffic overhead and because the client will likely request payload sizes that either fits its receive buffer or map nicely to their FlashROM layout.

Once the device has successfully flashed the new image or has determined that it is already up to date, it will call close on the bus object path and leave the session.

Firmware Image provider to update broker

Any number of protocols may be employed by a firmware image provider that an update broker could potentially communicate with. Rather than try to enumerate all the possible protocols, the update broker should provide plugin hooks to allow for the creation of plugins that will communicate with firmware providers using their preferred protocol.

For an update broker to provide a plugin system, the problem set needs to be abstracted. As with communications between the broker and device, there needs to be image discovery and bulk data transfer.

Because the problem set is exactly the same as it is for communication between the broker and device, it is possible to use the exact same AllJoyn protocol defined in this document between the firmware image provider and the update broker. This capability, actually enables a number of interesting possibilities that are mentioned elsewhere in this document.

One downside with using AllJoyn for the firmware image provider to update broker communications is that AllJoyn messages have significant overhead. Even with using the largest payload size possible, the overhead could be up to 0.2%. A very simple HTTP (or rather HTTPS) based protocol could be employed as well which would greatly reduce the overhead. This could be important when delivering firmware over a metered Internet connection. (When using HTTPS, the only overhead is the initial setup of the SSL socket, after that, the bandwidth usage is the same as for HTTP.) The rest of this section will describe the simple HTTP based protocol.

Image discovery

The layout of firmware images will be a simple collection of directories where the firmware image SHA256 hash is the name of the directory using ASCII characters 0-9A-F. The file structure would look something like the following:

- <*base URL*>/firmare/<*sha256 id*>/image – firmware image
- <*base URL*>/firmare/<*sha256 id*>/signature – cryptographic signature of image
- <*base URL*>/firmare/<*sha256 id*>/update – text file with the SHA256 ID of the update image – if it does not exist, there is no update.

Since the update broker knows the SHA256 hash of the currently running image on the device to be updated, it can send a GET request for the *<current sha256 id>/update* file. If the GET is successful, it

can then use the new SHA256 ID in that file to GET the <*new sha256 id*>/image and <*new sha256 id*>/signature files.

Image data transfer

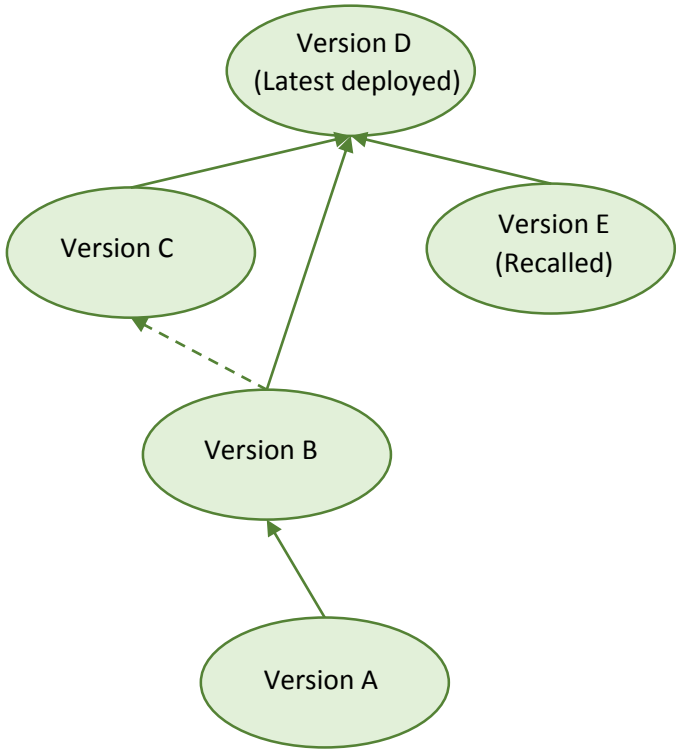
After issuing the GET for the <*new sha256 id*>/image file, the update broker just reads the bytes in from the network socket and saves them to a file. The update broker will use the information in <*new sha256 id*>/signature to validate the downloaded image.

Firmware identification and chaining

As mentioned elsewhere, a SHA256 hash of the firmware image will be used as the sole identifier for firmware images. This is far simpler to manage than any other aggregation of firmware version number, firmware build date, firmware build configuration, OEM personalization, hardware revision, model number, manufacturer, and any other bit information that could possibly impact selection of update firmware image.

The SHA256 hash is universally unique across all possible firmware images. Given that the hash is 256 bits in size, the odds that 2 different firmware images have the same hash are approximately 1 in 10^{77}. By comparison, large jackpot, multi-state lotteries typically have odds in the neighborhood of 1 in 10^{9}.

Chaining refers to the chain of firmware images that lead to the latest firmware image. In many cases, this looks just like a nice simple linear chain of firmware image versions. In other cases, however, could be more complex. This most often happens when the current firmware on a device is out of date by several revisions. It may be possible to update that device to the very latest version and skip all the intervening versions. The diagram below show a possible chaining.



Let's say we deploy a device that is running Version A and the latest is Version C. The device would need to update to Version B prior to updating to Version C (perhaps there was a change to the configuration format that needs to be converted). Now, let's say we have a device running Version B and the latest is Version D. This time Version B can be updated directly to Version D without having to install Version C. In fact, the upgrade path from Version B to Version C is replaced by the path to Version D.

So what happens if Version E is released and some devices update to it but it is later found to be flawed and the decision is made to roll back to Version D? This would involve altering the version chain such that Version D is made the successor to Version E.

As can be seen, the upgrade "chain" is really a tree. All firmware providers and update brokers must be able to keep track of the tree nodes so that when a device with an older version is discovered, it can be provided with the correct update image. It is also important to note that not only can new roots be added to the tree but that old versions can be made back into the root to accommodate rolling back to previous version.

In a corporate environment or even a development environment, it may be desirable to have different versions of firmware running on different otherwise identical devices. This can be accomplished by implementing a more sophisticated update broker that takes into consideration other information in the About message, such as the Device ID field and maintains an independent upgrade tree for specific devices that match that additional criteria. (Note, this capability should **not** be made available to update brokers for home environments as it will only cause problems for ordinary users.)

Implementation ideas and considerations

Update broker

Update brokers should be designed with the possibility that multiple update brokers could be installed on an AllJoyn network at a time and that each may cache the same update image. The retry mechanism described previously along with devices only accepting a single session on the update session port can provide a rudimentary load balancing mechanism provided the update brokers limit the number of outgoing sessions they setup at a given time.

To build further on the load balancing idea, it should be possible to develop an update broker that can download firmware images from other brokers by acting like an updatable device. That means the update broker would list all the SHA256 values it looks for in its own About message. This would cause other update brokers on the AllJoyn network to provide this update broker with a new firmware image when one becomes available.

Since most update brokers will be communicating with upstream image providers over the Internet, any number of protocols could be used to query for updates as well as download them. In order to provide generic support, the update broker should provide plugin support for firmware discovery and downloading,

Device

Given the nature of how the image discovery works on AllJoyn networks, it is possible for devices themselves to also act as limited update brokers. They would obviously be limited to only providing

updates to the firmware they are currently running. All they would need to do is keep track of the SHA256 hashes the identify firmware that can be updated to their own firmware image. If they detect another device on the AllJoyn network with a firmware ID that matches one of its prior versions, the device could then establish an update session as a normal update broker would. This capability may not be such a good idea for devices that get installed into a corporate environment where the IT department will want to vet updates before deploying them.

Firmware image provider

Security

Cryptographic signing of firmware images

Distribution of public verification keys

Revocation and replacement of compromised keys

Interfaces

org.alljoyn.BulkDataTransfer.FirmwareSelection

This interface gets implemented on update brokers. It provides the signal used to notify devices of new firmware updates as well as selection of firmware to download.

Properties:

None.

Methods:

OpenUpdate(*firmwareId*) -> *busObjectPath*

> This creates a new bus object path from which the device can download the updated firmware image. If the *firmwareId* is not known then the error org.alljoyn.FirmwareSelection.Error.NotFound will be returned instead. If the session this call was made on is lost prior to CloseUpdate() being called, the update broker will treat that as if CloseUpdate() had been called. If multiple devices call OpenUpdate() with the same *firmwareId*, then the update broker will return the same bus object path and reference count its usage so that the bus object can be removed when all devices are done. It is permissible for a device to open multiple different update images at the same time or sequentially in the same session. This is to support devices that may have multiple sub-systems where each sub-system has its own firmware image.
>
> *firmwareId*: byte[] – Firmware ID of the currently running image that needs to be updated.

busObjectPath: bus object path – Bus object representing the firmware image to be downloaded.

CloseUpdate(*busObjectPath*)

This lets the update broker know that a device no longer needs access to the update firmware image. It is preferable to call this prior to leaving the session. If *busObjectPath* is invalid, it will be silently ignored. The update broker will keep track of which firmware images were opened for a given session and only close those that were opened in the session.

busObjectPath: bus object path – Bus object representing the firmware image that was downloaded.

Signals:

UpdateAvailable(*firmwareId*)

This signal informs devices that an updated firmware image is available for the indicated *firmwareId*. The *firmwareId* is provided in the event that a device actually has multiple sub-systems with different firmware images. Note that the firmware ID not the new firmware image; it is the firmware ID of the currently running image for which there is an update available. If it were the ID of the new firmware, then any device that has multiple firmware images to manage would not know which one has an update available.

org.alljoyn.BulkDataTransfer.Information

This interface is implemented on bus objects that represent a file or some other type of fixed size, contiguous block of data that can be uploaded to or downloaded from.

Properties:

Size: uint64 – Current size of the file or data block.

Sha256Hash: byte[] – The SHA256 hash of the file or data block. The length of the byte array shall always be 32 bytes.

Methods:

None.

Signals:

None.

org.alljoyn.BulkDataTransfer.Get

This interface is implemented on bus objects that represent a file or some other type of fixed size, contiguous block of data that can be downloaded from.

Properties:

None.

Methods:

Get(*startOffset*, *size*) -> *dataSegment*

This gets a segment of data. The size of *dataSegment* must never exceed that specified in *size*. If *startOffest* is beyond the end of the file or block of data, then the error org.alljoyn.BulkDataTransfer.Error.EndOfFile will be returned.

startOffset: uint64 – Start position in the file or data block to retrieve.

size: uint32 – Maximum number of bytes to include in the returned data segment.

dataSegment: byte[] – The data in the requested segment.

Signals:

None.

org.alljoyn.BulkDataTransfer.Put

This interface will not be implemented by the update broker. It is defined here for completeness of a generic bulk data transfer interface definition. This interface is implemented on bus objects that represent a file or some other type of fixed size, contiguous block of data that can be uploaded to.

Properties:

None.

Methods:

Put(*startOffset*, *dataSegment*) -> *sizeWritten*

This puts a segment of data. If *startOffset* is at the end of the file or data block, the data in *dataSegment* will be appended to the file or data block. If *startOffset* is before the end of the file or data block, the data in *dataSegement* will overwrite what was previously there. If *startOffset* is past the end of the file or data block, the error org.alljoyn.BulkDataTransfer.Error.EndOfFile will be returned. If there is some error storing the data in *dataSegment*, the error org.alljoyn.BulkDataTransfer.PutFailed will be returned.

startOffset: uint64 – Start position in the file or data block to put.

dataSegment: byte[] – The data to be written.

sizeWritten: uint32 – The number of bytes actually written. This may be less than the size of the *dataSegment* sent in which case the sender would presumably retry sending the remaining data.

Signals:

None.