Remote Vehicle Interaction

Architecture High Level Description

RVI Release v1.0

Ulf Wiger
uwiger@jaguarlandrover.com

Tatiana Jamison
tjamison@jaguarlandrover.com

Magnus Feuer
mfeuer1@jaguarlandrover.com

Revisions

Revision	Date	Author	Notes
Draft 1	2014-06-11	Magnus Feuer	First Draft.
Draft 2	2014-06-11	Magnus Feuer	Review feedback from Arthur Taylor integrated.
Draft 3	2014-06-12	Magnus Feuer	Additional feedback from Arthur Taylor and Paul Hanchett integrated.
Draft 4	2014-06-14	Magnus Feuer	Formatting and minor corrections
Draft 5	2014-06-26	Magnus Feuer	Renamed document, removing the Tizen reference. Updates from security review: 1. Change document license to Creative Commons. 2. Remove all security between Service Edge and services Securing the service - Service Edge link security is now scoped out to implementation / deployment / ops. 3. Add certificate timestamp field. Allows chronological sorting of certificates 4. Add node detection technique examples 5. Add per-user service access to the Issue list 6. Add suggestion on how to avoid node spoofing to issue list
A	2014-09-22	Magnus Feuer	Updated to reflect implementation of RVI v0.2. 1. Renamed Store and Forward component to Schedule. 2. Topic tree renamed service name 3. Use case in chapter 8 rearranged to reflect v0.2
pB1	2016-11-10	Ulf Wiger	Updated to reflect implementation of RVI v0.5 and planned for v1.0
B	2017-01-30	Ulf Wiger	Updated to reflect implementation of RVI 1.0.

1. Table of Contents

2. References

[1]	Google Protocol Buffers	https://code.google.com/p/protobuf/
[2]	Requirement Specification	TBD

3. Acronyms and definitions

RVI	Remote Vehicle Interface
SOTA	Software Over The Air
TSP	Telematics Service Provider
RVAM	Remote Vehicle Access Manager
Node	A vehicle, mobile device, or backend server running RVI-integrated services
Component	An internal service inside the RVI system
Manager	A higher level component
Service	An external application connected to the RVI system
Service Name	A global registered name used to identify and locate a specific service.
Network Address	An URL, IP address, MSISDN, or similar that a node can be reached at
JWT	JSON Web Token

4. Introduction and Purpose

This document introduces the Remote Vehicle Interaction (RVI) architecture on a high level, outlining its design, the components, and the core use cases. It is intended for the reader who wants a comprehensive overview of RVI and its inner workings.

The purpose of the RVI architecture is to enable vehicles, mobile devices, and backend servers to exchange today's and tomorrow's connected car services in a robust, secure, and versatile manner. RVI will place a minimum number of requirements and restrictions of the services that use it, aiming at giving applications a maximum degree of freedom in their implementation. Examples of applications using RVI are remote door unlock, software upgrades (through the dealer, OTA and USB sticks), climate control from mobile device, syncing of media files between the cloud and the vehicle, geo fencing, etc.

The RVI architecture describes a set of components and services that form a distributed, sparsely connected, secure P2P network where vehicles, mobile devices, and backend servers can access each other.

The complete set of RVI architecture documents will be used as the basis for a reference open source implementation of RVI that will be released to the community.

The philosophy behind the RVI effort is that the reference implementation drives the specification, and that any design issues resolved in the implementation flows down to the specification. Thus, if the implementation and the specification are in conflict with each other, the implementation takes priority. The feedback from the implementation will be used to refine the RVI architecture and specification itself.

4.1. Document Structure and Reader Assumptions

The RVI architecture is described in the following hierarchy of documents:

1. **High Level Description (HLD) (this document)**
   Goes through the overall schematics, its included components, and core use cases.

2. **RVI Requirements Specification**
   Contains detailed requirements derived from, and explained in this document.

It is recommended that the documents are read in the order given above.

The reader of this HLD is assumed to be familiar with the following areas:

1. **Mobile terminology**
   Terms such as 3G, handset, pppd, and mobile data links will be used throughout the HLD.
2. **Automotive applications**
   Media Players, door lock management, and other applications will be used as examples.
3. **Networking**
   Distributed system terminology, IP-terms, and web technologies such as JSON-RPC and HTTP will be used.

4.2. Issues

This specification is not complete. Below is a list with currently identified issues and shortcomings that should be addressed.

4.2.1. Data Link node discovery

There is no specification for how two nodes should find each other prior to running the authorize/announce use case. While this is technically in the domain of each Data Link implementation, best practices such as UDP broadcasts, connection attempt to well-known addresses, etc., should be documented, as well as related security implications.

5. Requirements and Objectives

There are several objectives for the design of RVI and its reference implementation.

The core mission is to provide a specification and implementation that is easy to adopt for vendors and customers, who are either are starting from a blank page or want to integrate their existing solution into the RVI framework.

The following chapters outline the high level requirements and objectives being pursued.

5.1. Internet Connectivity Optional

If two nodes see each other over a communication link, their services must still be able to exchange traffic even if neither node has an Internet connection. The typical use case is an owner who wants to unlock and start a car in a garage using his/her mobile phone, and wants to do this even if there is no carrier signal. The access app on the mobile phone must be able to talk to the access service in the vehicle without having to authenticate toward a (non-reachable) central server.

5.2. Secure Communication

RVI must provide protection from eavesdropping, impersonation, privilege abuse, etc.

5.3. Zero Service Configuration

Given the Internet requirement above, a new service, be it a mobile phone app, an in-vehicle HVAC controller, or a cloud-based traffic information service, must be able to register themselves without updating a central repository. A new service should only have to present correctly signed requests to the RVI system in order to register and access other, remote services.

Please note that nodes (hosting services) still need to be provisioned with addresses, protocols, and access rights, all expressed through certificates and credentials.

5.4. Decentralized Service Discovery

Nodes need to discover services on other nodes without involving a central repository so that two nodes without an Internet connection can explore each other's available services. This is also true for when two nodes communicate with each other over non-IP based links such as Bluetooth, IR, or even USB sticks (software updates or content transfer).

5.5. Self-Carried Authorization

When two nodes have discovered each other's services, they need to authenticate their right to access them. With the optional internet connectivity requirement, these authorizations have to be made in a pure peer-to-peer fashion without third party involvement.

5.6. Interoperability

An organization shall be able to implement individual components and drop them into an existing set of RVI components without compatibility issues, given that they use the API protocol (JSON-RPC, etc.)

5.7. Connectivity Awareness

Both Services and individual RVI components need to know if a remote node and its services can currently be reached or not. This spans from a mobile device knowing if a peer-to-peer connection for a specific vehicle is available or not, to a cloud-based media server wanting to know the available bandwidth to a media player, to a vehicle wanting to know its currently available data channels.

5.8. Sparse Connectivity Support

Since data links come and go, often with short notice, and services on two nodes may not see each other for weeks at the time, RVI shall handle resend attempts, delivery validation, timeouts, and store & forward for transactions in order to support sparse connectivity.

5.9. Hybrid Deployment Support

RVI nodes will have to integrate with servers and vehicles using other designs and products. An RVI deployment must, with custom protocol implementations, be able to connect and interact with other telematics and M2M systems.

6. Architecture Overview

Below is a layout of a typical RVI deployment, where the blue and green components form the core of the architecture.

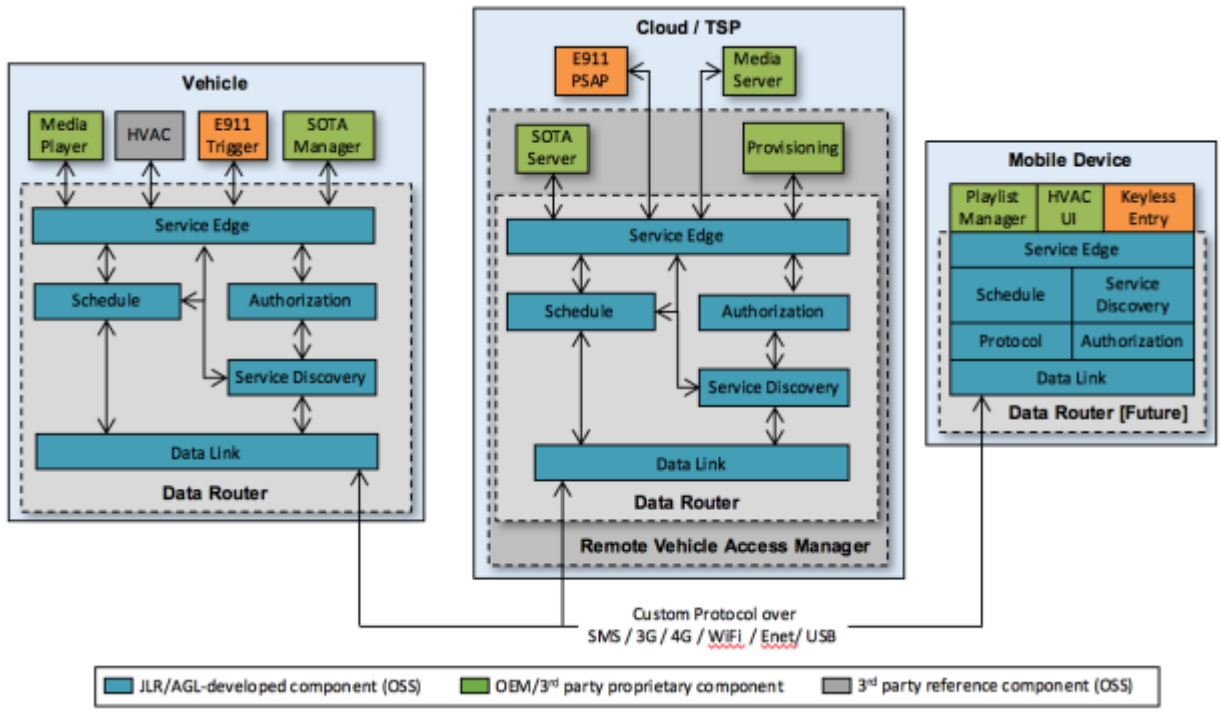

Figure 6-1 – Architecture Overview

Services can communicate peer-to-peer with each other, allowing, for example, a mobile device to interact directly with a vehicle, even if neither have an Internet connection to a Remote Access Vehicle Manager.

Not visible in the architecture diagram is the fundamental decision to base RVI on the TLS 2.0 security model, X.509 certificates and public/private encryption and signing. The implications of this will be described in more detail in relevant chapters. As TLS-based security is a broad and moving target, this document attempts to define as narrowly as possible which TLS aspects are used, and how. How to ensure safe handling of certificates and, especially, the certificate authorities used for signing, is up to each organization implementing RVI-based operations.

The components are described in the following chapters.

6.1. Data Router

A high-level aggregate of lower-level components that delivers transactions between services robustly and securely over sparsely connected networks. Different compositions of the data router execute inside the vehicle, on the backend server, and in mobile devices.

6.2. Remote Vehicle Access Manager (RVAM)

The server-side RVAM aggregates the Data Router, Billing & Charging, Software Over The Air (SOTA), and Provisioning into a single server system that extends the service transaction routing with external device and software management, as well as billing, charging, and reconciliation.

6.3. Service Edge

Service Edge, present inside the Data Router, acts as a service-facing API coordinating all traffic sent to and from local services. It is responsible for authorizing and routing requests to their targeted end points.

6.4. Schedule

Schedule receives requests from Service Edge, and, in case the addressed service cannot be reached, stores the request until a data link to the service becomes available. It is also responsible for management communication channels to other nodes (and their services) through Data Link.

6.5. Data Link

Responsible for bringing up and tearing down data channels to remote nodes. Typical data links are WiFi, SMS, 3G, 4G (over PPP), Ethernet, and USB sticks containing requests. Data Link can be ordered by Store and Forward to setup or disconnect a data link, and will report to subscribing parties when link to a remote node becomes available or unavailable.

6.6. Service Discovery

Service Discovery tracks service availability on local and remote nodes. The Service Edge queries Service Discovery in order to retrieve a target node address for a specific request. Service Discovery can also relay service availability announcements to neighboring nodes with appropriate access privileges.

6.7. Authorization Manager

The authorization manager adds necessary signatures and certificates to outgoing transactions to remote nodes, allowing requests to be validated prior to execution by the targeted services. Authorization will also validate incoming requests from remote services before Service Edge sends them on to their local service destinations.

6.8. Provisioning Server

The Provisioning server, a part of the RVAM, supports Service Discovery with information about nodes (mobile devices, vehicles, and servers) available in a RVI network, and how they can be reached. Provisioning also supports the Authorization Service with certificates used to sign outgoing requests to prove their authenticity. In many instances, Provisioning is a gateway to one or more external provisioning services hosted by TSPs, Enterprise IT, and other organizations.

6.9. Software Over The Air (SOTA)

Software Over The Air is, from a strict RVI perspective, a generic service with no special access to the internal RVAM / Data Router components. Since SOTA (and Firmware Over The Air) is so central to RVI, it has been included in the architecture as a specification and reference implementation. The SOTA server, however, can be replaced without modifying any other components apart from its SOTA Manager counterpart on the vehicle.

6.10. Mobile Device Data Router [Future]

Mobile devices, with their often constrained execution environment, can sometimes not be suitable for having a loosely coupled set of components forming the Data Router. Instead, these targets may need a monolithic service or library that presents a similar interface

7. RVI-Wide Concepts

7.1. Messages

The RVI inter-node protocol consists of the following messages, introduced here with references to more detailed descriptions. Detailed descriptions can be found in chapter **Error! Reference source not found.**.

REQ-RVI-GEN-1	Support all core RVI messages
Mandatory	*Fragmentation messages must at least be recognized*

7.1.1. Authorization (“au”)

Initial handshake message, exchanging access credentials and setting data link encoding

7.1.2. Service Announcement (“sa”)

Informs neighboring nodes about the availability of services

7.1.3. Configuration Change (“cfg”)

Changes a characteristic of the present connection, e.g. the encoding.

7.1.4. Service Invocation (“rcv”)

Calls a service either on the local node or a remote node.

7.1.5. Ping (“ping”)

Heartbeat message.

7.1.6. Fragmentation messages (“frg”, “frg-get”, “frg-end”, “frg-err”)

Allows large messages to be split into smaller fragments and transmitted reliably. See Chapter 16.

7.2. Service Invocations

An RVI service or client can send requests to available services using the ‘recv’ message type. Requests are asynchronous by default, but can optionally be made synchronous.

7.3. Service Names

A service, connected to an RVI node, is identified by a service name that is globally unique across all RVI nodes in a service name tree, meaning that a specific service name resolves globally to the same physical service instance for all nodes.

The syntax of a service name is:

```
[domain]/[device type]/[UUID]/[service name]
```

.Figure 7-1 – Service name syntax

The components of the entry are as follows:

1. **[domain]**
   A domain name describing the organization that hosts a sub-section of the root. All service paths under the domain name are managed by the given organization.

2. **`[device type]`**
   Specifies the type of device in question. This is primarily of documentary value.
3. **[UUID]**
   A universally unique identifier (see RFC 4122[1]). UUIDs are regarded as unique for practical purposes, especially if using a high-resolution clock and an IEEE 802.3 MAC address for node id. The 'owner' of each domain needs to ensure that the UUID is unique within the given domain.
4. **`[service name]`**
   A path describing the given service hosted by the organization. The path should be a hierarchical identifier going from more general to more specific. Name matching is case-insensitive, but it is strongly advised to stick to a single case, e.g. lowercase, throughout.

Below is an example of a service name, with numbered components.

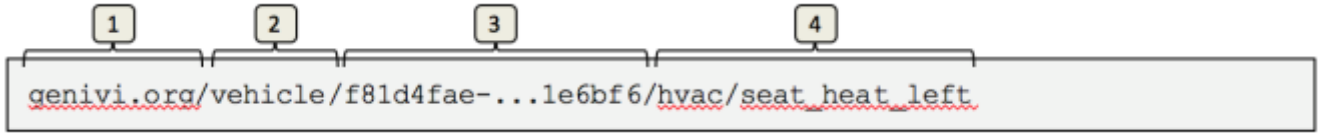

Figure 7-2 – Typical service entry

Please note that components 1, 2 and 3 are standardized. The [service name] component can be anything that complies with the MQTT topic name specification, albeit with the following additional restrictions on the Fully Qualified Service Name (FQSN):

- The domain portion must conform to RFC1035[2]
- Matching of the FQSN is case-***in***sensitive

[1] https://tools.ietf.org/html/rfc4122
[2] https://tools.ietf.org/html/rfc1035

- FQSN length is limited to 2048 bytes
- No leading forward slashes
- At least 4 topic levels are required (domain, device_type, device_uuid, followed by at least one more level)
- RVI reserves “RVI” as a special sublevel following the uuid (note, however that name matching is case-insensitive). This is used for e.g. credential management services (see chapter 10)
- Trailing wildcard matching with '#' only
- Single topic matching with '+'
- Any characters acceptable except +, #, and null
- Each topic level must be at least one character long

Since a service name is system-wide unique, i.e. a service name resolves to the same specific service for all nodes in a network, the identity of a node that hosts the service is embedded into the service name.

When a request is to be sent to the service using a service name, the service name will be resolved to a network address, which will then receive the request. See chapter “6.6” for a description of how a service name is translated to a network address.

7.3.1. **Internal service names**

RVI can create internal service points, e.g. to receive replies to synchronous requests. For security reasons, these service names must be node-specific, unique and not possible to call from the outside. Internal service names begin with the node service prefix preceded by a dollar sign (‘$’). For example, the node id “genivi.org/vehicle/f81d4fae-...1e6bf6” would correspond to the internal service name prefix “$genivi.org/vehicle/f81d4fae-...1e6bf6”.

The Service Edge must reject any invocation attempt from the outside addressed to any service name starting with ‘$’.

7.4. Service Name Matching

Access control in RVI is managed using self-carried privileges, associated to a particular node, contained in a JWT. Each JWT contains the device’s X.509 certificate (to match the one presented in a TLS connection) and is signed by the Provisioning Server (a trusted Certification Authority) to ensure integrity.
In addition, the JWT contains a list of rights granted to that node. Each entry in that list follows the service name rules specified in 6.4 (Service Names), with the following modifications:

1. **Single-Level Wildcard ('+')**
   A single topic level can be wildcarded using the '+' character. The single-level wildcard **must** be bracketed by the topic divider '/' on either side. The following are legal patterns:

   genivi.org/vehicle/+/control/#
   genivi.org/+/+/media/#

   The single-level wildcard '+' can be used repeatedly in the same pattern. Each use corresponds to one topic level.

   The following **are not** legal patterns:
   genivi+/vehicle/#
   +.org/#

2. **Multi-Level Wildcard ('#')**
   Patterns can include wildcards that match across an arbitrary number of service levels using the '#' character. The multi-level wildcard may only be used at the end of a pattern, and must be the sole character specified in that topic.

   The following are legal patterns:

   genivi.org/#
   genivi.org/vehicle/+/control/#

   The following **are not** legal patterns:

   #/control
   genivi.org/vehicle#

The details and use of privilege tokens are described in detail in "9 Node and Service Provisioning."

7.5. Node addressing

Nodes, setup in the Provisioning Server, are servers, vehicles, devices running the RVI and a number of services. Each node can communicate with one or more other nodes over various data links.

A service name is resolved to a network address by Service Edge, which will keep track in real time of how to reach all services known to it.

Since it is up to Service Discovery to convert a service name to a network address, the exact procedure of converting a service name to a network address is implementation dependent.

Figure 7-3 below shows the high-level call flow for address resolving a service, identified by its name, to a network address.

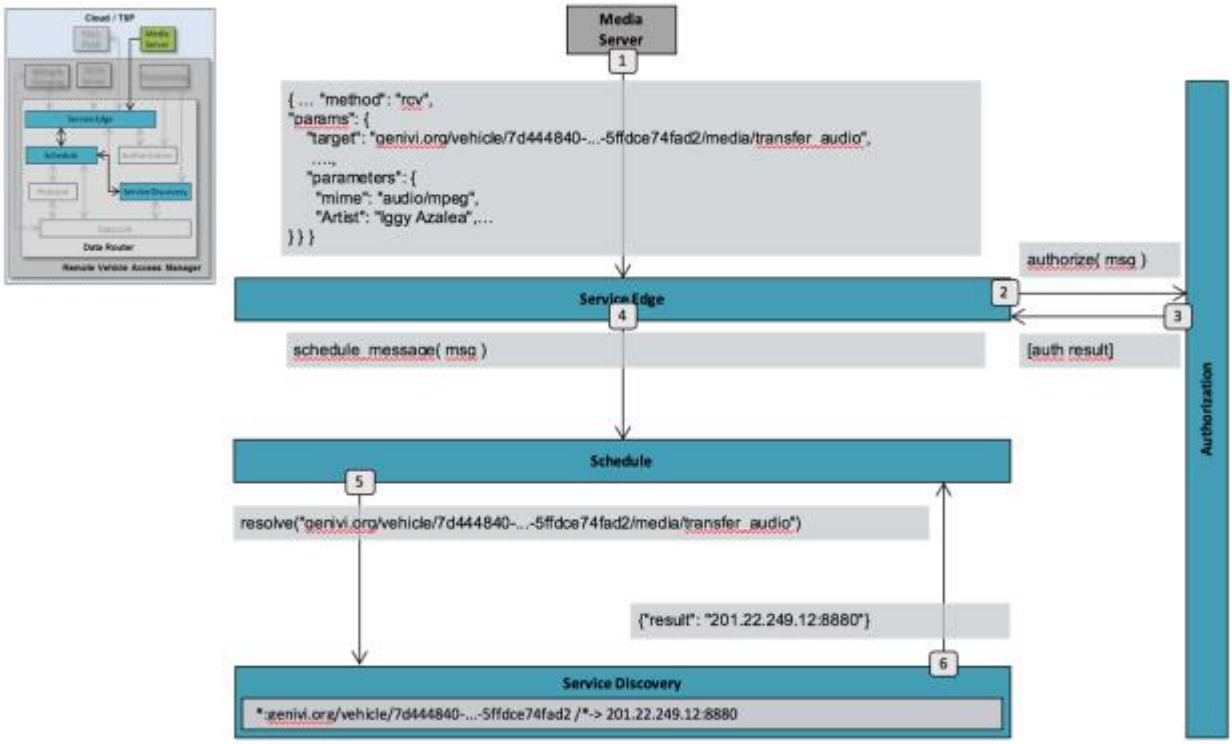

Figure 7-3 – Node-to-network address resolution

The steps above are as follows.

1. **Send Request service edge**
   The Media Server sends a request, to be forwarded to the vehicle with UUID 7d444840-9dc0-11d1-b245-5ffdce74fad2, to Service Edge.

2. **Authorize request**
   The Service Edge checks with Authorization whether the client is allowed to call the service in question.

3. **Return authorization result**
   If the initial authorization check fails, an error response is sent to the client. Otherwise, processing continues.

4. **Forward to Scheduler**
   Service Edge forwards the request to Scheduler, which will manage data link setup and data transmission.

5. **Lookup node address**
   Scheduler sends a request to Service Discovery, providing the full service name, as received from Media Server, in order to retrieve the network address of the node (vehicle) that hosts the targeted service.

6. **Return network address**
   Service Discovery, having matched the service name against its internal database, returns the network address, which in this case is a IP/Port pair, to Scheduler.

Please note that this is a simplified scenario since no data link types (3G, WiFi, SMS, etc), specifying the recommended delivery methods to the destination node, is returned by Service Discovery.

7.6. Data Link Encoding

At the highest level, RVI messages can be represented as JSON messages. When being prepared for transmission, they can be encoded using any compatible encoding supported and agreed upon by both sides. Currently, RVI specifies two encodings: JSON and msgpack.

7.6.1. **JSON Encoding**

This encoding is trivial as it perfectly matches the abstract representation.

7.6.2. **Msgpack Encoding**

The msgpack encoding system is designed to support efficient encoding of JSON terms. However, a few details need to be determined:

- All strings (including keys) are encoded as msgpack binaries (utf8 encoding)
- Integers are limited to 64-bit integers

7.7. RVI Security Scope

Security, in the context of this architecture, is the process of authenticating services and authorizing them to access other services in an RVI system. A service should only be able to discover and access other services it has received authorization to from a provisioning system trusted by both the sending and receiving RVI nodes.

The constraints imposed by the sparse connectivity requirement means that each node, and its services, must be able to authenticate themselves toward other nodes without having to rely on a connection to a central provisioning server.

The security management of the RVI architecture covers the following areas:

1. **Certificate Generation**
   The provisioning server must have support for generating and/or signing X.509 certificates for each RVI node. When connecting, each node validates the other node's certificate and its trust chain. The certificates are also included in the generated credentials (see below) to ensure that they all represent the same entity.

2. **Credential Generation**
   The provisioning server has support for generating credentials granting access for a given node to one or more other services. See chapters 8.2 and 8.5 for details.
3. **Credential Distribution**
   Once a credential has been generated by Provision Service, it has to be sent out to the node that the certificate was generated for. This transmission can be implemented through an RVI built-in credential service via the RVI protocol.

4. **Certificate Revocation**
   All certificates and credentials are specified with a time interval within which they are active. Sometimes, however, they need to be revoked prior to their expiration date. Such revocation can also be implemented through RVI services.

The security management of the RVI does **not** cover the following areas.

1. **Provisioning Server Security**
   The Provisioning Server's method for validating users and determining which credentials to issue are beyond the scope of this document. The function of the Provisioning Server is described only so far as it relates to the interaction between the RVI node or user and the Provisioning Server.

2. **Node – Certificate linking**
   Since the RVI will operate on many devices, platforms, and network protocols, there is no uniform way of linking a certificate to a specific device using a mac address, CPU ID, etc. As a consequence, there is no way for a node receiving a certificate to validate that the sending device is actually the owner of that certificate. If a device manages to steal a certificate, the private key and credentials from a node, the device will then be able to impersonate the node and hijack its traffic.

3. **Node protection of certificates**
   Another consequence of the unknown devices and platforms that will run RVI, is that the storage and protection of certificates and key pairs on a node is outside the scope of the RVI design and has to be addressed by the implementation of individual Authorization components.

4. **Service – Service Edge Authentication and Authorization**
   Services are implicitly trusted by the Service Edge and are expected to execute inside the trusted domain of a node. It is up to the implementation, deployment, and operations of an RVI system to ensure that no illicit services gain access to Service Edge.

8. Service Management

Service Management consists of a set of components, use cases and APIs that enables services to:

1. **Register with RVI**
   A service can connect to RVI (through Service Edge) and register itself.

2. **Announce themselves**
   A registered service can announce its availability not only to its current node, but also to other, remote nodes in the RVI network.

3. **Discover other services**
   A service can receive availability announcements from other services, with all information necessary to connect and send requests to them.

4. **Invoke other services**
   Once authorized, a service can send requests to other, remote services, having them carry out tasks and send back the result.

The following chapters describe the service management on a high level.

8.1. Service Registration

A Service can register itself with RVI through Service Edge to set itself up for sending and receiving requests, as is shown below.

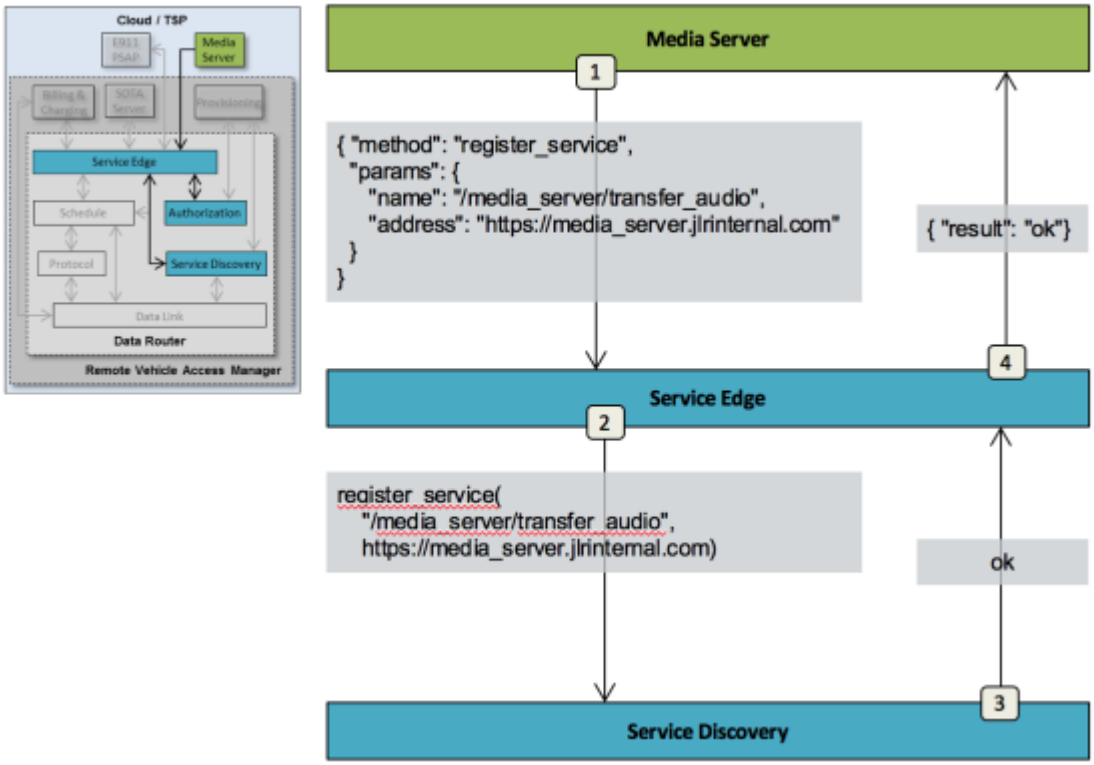

Figure 8-1 – Service authorization

The steps above are as follows.

8.1.1. Step 1 - Register with Service Edge

The Media Server sends a request to register itself with Service Edge.
The request contains the following elements:

1. **Service Name**
   The service name to be registered with the node.

2. **Service Address**
   Specifies where to forward requests and replies targeting the service.

8.1.2. Step 2 - Register with Service Discovery

The service is forwarded by Service Edge to Service Discovery, who will announce it to all matching subscribers. After this point, the service will be forwarded requests addressed to it.

8.1.3. Step 3- Confirm Service Discovery registration

Service Discovery confirms that the service has been registered and announced.

8.1.4. Step 4 - Confirm Service Edge registration

Service Edge replies to Media Server service that the registration was successful, and that traffic is to be expected.

8.2. Credentials

Credentials are, in the RVI context, cryptographically signed JSON structures that prove the authenticity and authorization of one node to another.

Credentials are exchanged, peer-to-peer, between two nodes, and must be thoroughly parsed and validated by the receiving node. A faulty credential MUST result in the connection being terminated.

An example of a credential is given below (with non JSON-compliant comments added for clarity):

```
{
  // The credential owner's PEM-encoded X.509 certificate
  "device_cert": "MIIB8zCC...kYtVbYs=",
  // System-wide global id for the credential
  "jti": "f81a1653-89bd-4be7-924e-758dc7861f4f",
   // The issuing organization
  "iss": "genivi.org",
  // Service name prefixes that can be accessed by
  // the credential owner
  "right_to_invoke": [ "genivi.org/" ],
  // Service name prefixes for which the credential owner is allowed
  // to process requests
  "right_to_receive": [ "genivi.org/" ],
  // Optional authority delegation
  "right_to_delegate": [ {"right_to_invoke": ..., "right_to_receive": ...} ],
  // Identity of delegate credential used to sign this credential
  "delegate": "e75b2354-...53654",
  // UTC timestamp of when the credential was created
  "create_timestamp": 1448683742,
  // UTC timestamps of when the credential starts/stops being valid
  "validity": {
      "start": 1448683742,
      "stop": 1480219742
    }
}
```

Figure 8-2 – Credential example

The following elements are included:

1. **sub (“Subject”)**
   The identifier for the subject of the privilege token, which is expected to be the PEM-encoded X.509 certificate of the credential owner, included in the credential to ensure that the credential owner and the certificate owner are the same.

2. **jti (JWT ID)**
   A system-wide unique identity of the credential. The responsibility for uniqueness lies with the Provisioning Server. Credentials should be regarded as Write-Once, in the sense that any updates should be made in the form of a new credential, revoking the old one. This is to avoid having multiple versions of the same credential floating around in the system.

3. **iss**
   The issuing organization. This item is not used by the core RVI system.

4. **right_to_invoke**
   The destination services which the credential authorizes the source service to access. Specified as one or more service name prefixes, optionally using MQTT-style single-level wildcards ('+'). During request authorization, the request is pattern- and prefix matched against the destination fields. This list will be installed in Authorization of credential-receiving nodes to be matched against requests received. It is also used by the credential owner to validate outgoing requests before passing them on. The 'right_to_invoke' option should be omitted if 'delegates' is present.

5. **right_to_receive**
   A list of service name prefixes for which the credential owner is authorized to process requests. This list will be installed in Authorization of credential-receiving nodes to be matched against outgoing requests. Even though nodes must reject requests that don't match any of the 'right_to_receive' lists in their own credentials, other nodes must take responsibility not to send such requests in the first place. Credential owners also use 'right_to_receive' lists to filter outgoing service announcements. A node must refrain from announcing as available a service for which it is not allowed to receive requests. The 'right_to_receive' option should be omitted if 'delegates' is present.

6. **delegates**
   A list of 'right_to_invoke' and 'right_to_receive' pattern pairs which identify service subsets for which the credential owner can issue credentials for other users. The 'delegates' option should be omitted if the 'right_to_invoke' and/or 'right_to_receive' options are present.

7. **Delegate**
   If present, identifies a credential used as for authority delegation. The delegate credential is expected to contain the PEM-encoded certificate ("device_cert") used to sign the current credential, as well as "right_to_delegate" instructions limiting the permissible scope of the "right_to_invoke" and "right_to_receive" patterns in the current credential.

8. **iat ("Issued At")**
   Specifies the time, in UTC, when the credential was created.

9. **"nbf" ("Not Before") and "exp" (Expiration Time)**
   The start and stop dates, specified as UTC, within which the credential is valid.

See chapter “Node and Service Provisioning” for details on how certificates are exchanged between nodes.

8.2.1. Credential Encoding

Credentials are encoded as JSON Web Tokens (JWT) – specifically as the sub-class JSON Web Signatures (JWS) with compact serialization, signed with the “RS256” algorithm using the private key of the Provisioning Server. Each node validates the signature using the corresponding public key.

```
BASE64URL(UTF8(JWS Protected Header)) || '.' ||
BASE64URL(JWS Payload) || '.' ||
BASE64URL(JWS Signature)
```

Figure 8-3 – JSON Web Token (JWT) structure

For efficient processing, key information elements from the credential should be duplicated as claims in the JWS Protected Header. Specifically, the following should be contained in the header:

- “jti” (JWT ID)
- “rviDel” (element “delegate”, if present)
- “exp” (Expiration Time) and “nbf” (Not Before)

Specifically, the “rviDel” claim identifies the identity of the delegating credential, which can be located via the “jti” element in the header. This allows for lazy processing of the JWTs, avoiding having to fully decode, parse and validate each JWT in order to find the right one.

8.3. Service Discovery

Service discovery is the process in where the availability of services is announced between two or more nodes. In order to fulfill the decentralized service discovery, sparse connectivity, and zero service configuration requirements stated in “”, a somewhat different strategy was chosen for this process. This strategy breaks down into the following two tenets.

1. **Peer to Peer Service Announcements**
   Since two nodes exchanging service availability information may not have an internet connection to a backend server, the exchange must be purely P2P without relying on a trusted third party for

validation.

2. **Pairing-based Service Discovery**
   While traditional service discovery mechanisms are often broadcast information over a local network, the sparse connectivity environment of RVI makes that impossible. Instead two nodes exchange information on an opportunistic basis as soon as they see each other. A node keeps track of all other nodes it has ever seen together with the services that were available on each of these nodes when they were last seen. When two nodes see each other again, they exchange updated service information.

A backend server/cloud node will see most of the other nodes as they connect in to the server and will, over time, form a complete picture of all other nodes (vehicles, mobile phones and other devices) and their available services. Services connected directly to the backend node will, through its Service Edge, have access to all other nodes and their services, as is shown below:

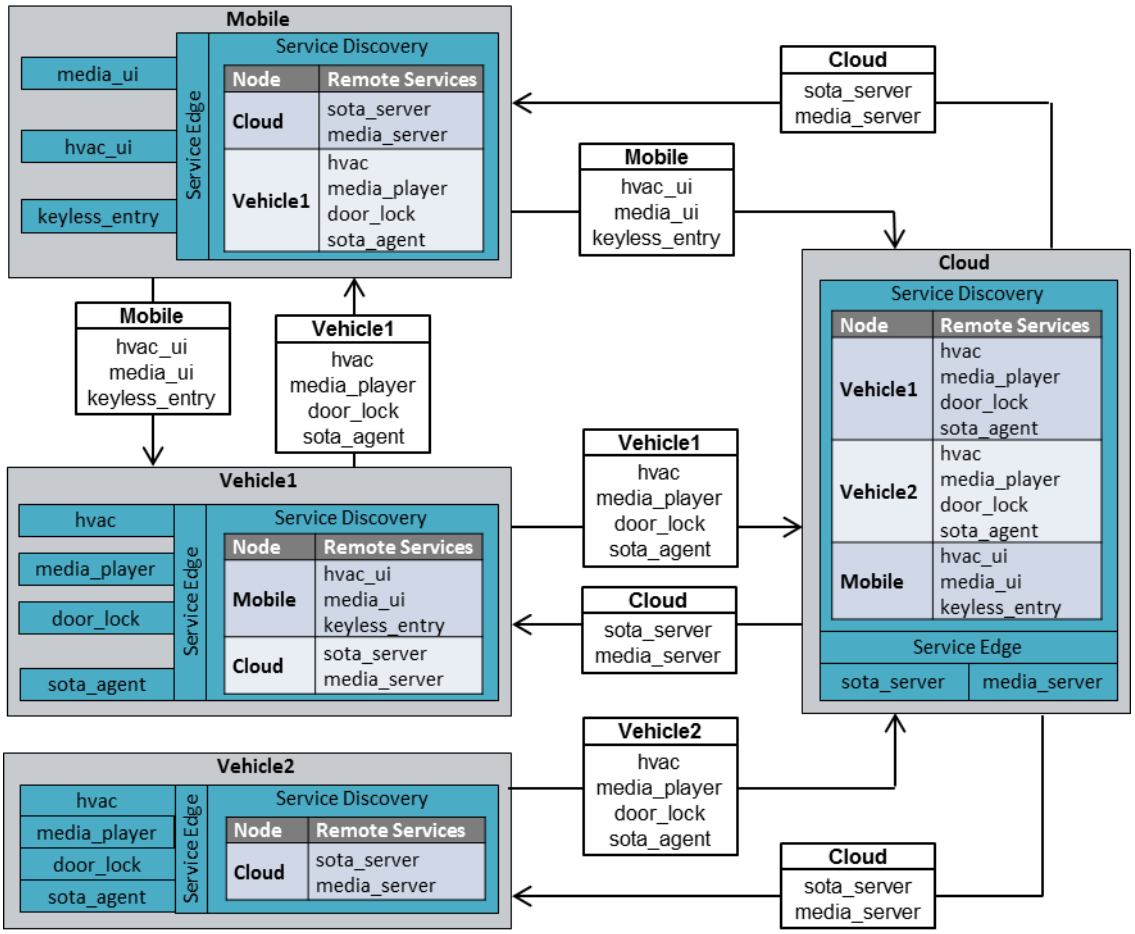

Figure 8-4 – Service Discovery mechanics

In the case above, Mobile and Vehicle1 exchange services directly with each other on sight, as well as with Cloud, while Vehicle2 only exchanges services with Cloud.

The end result is that Mobile, having access to Vehicle1's services can control its media player, HVAC, and door locks, whole at the same time having access to Cloud's services. Meanwhile Vehicle2, having never seen Mobile, can only access Cloud's services.

The Service Discovery process between two nodes is described below.

8.4. Node Detection

When two or more nodes establish a communication channel, they need to detect the presence of each other before they can execute the Service Discovery process.
The exact means of how this is done is wholly dependent on the mechanics of the underlying network. There are, however, a number of best practices that can be employed for standard network models, described in the following chapters.

8.4.1. **Vehicle connection to well-known server**

When a vehicle's Data Link is instructed to connect to a backend server, identified by a pre-provisioned URL or IP address, the Data Link can ping the server once the communication channel is up.
The ping, which is a TCP/IP connection, does not have to transmit any information and can immediately be disconnected. The receiving server will ask the operating system for the peer address of the TCP connection to determine that there is a node at a given address that wants to execute Service Discovery.

If the protocol used to drive the Service Discovery process between two Data Links is connection based, the TCP connection can be used in the subsequent authorization and announce steps.

8.4.2. **WiFi network connections**

When a Data Link connects to a WiFi, LAN or CAN network, there may be a number of other nodes available to execute the Service Discovery process toward.

In these cases, each connected Data Link can send out a UDP/IP broadcast or multicast packet to a well-known address and port that all Data Links listen to. The payload of the packet is irrelevant since the receiving Data Link can look at the peer address of the packet to retrieve the IP address of the remote Node, and then initiate the Service Discovery Process against that Node.

8.4.3. **P2P connections (Bluetooth, serial)**

In a strict P2P connection, where two Data Links are in direct connection with other, the Service Discovery process can be initiated directly toward the remote end.

8.4.4. **SMS**

SMS communication between a vehicle and a central backend server can be handled as a connection to a well-known server described in chapter 8.4.1.

SMS communication between two nodes requires the Data Link to know the target's MSISDN (phone number) since all mobile terminals are addressable at all times. There is no way for an SMS-connected Node to determine if another SMS-connected node is currently online or not. Instead the sending Node initiates its Service Discovery process by submitting an SMS to the network, and then time out of there is no reply from the target node.

8.5. Authorization

When two nodes see each other, either for the first time, or on a recurring connection, they start by sending authorization information to their counterpart. In order to maintain the pairing paradigm, each node will send an Authorization package to each other node it sees.

Although the RVI protocol is a peer-to-peer protocol, the handshake messages are initiated by the connecting node, or 'client'. The accepting node will be referred to as 'server'. If the transport protocol doesn't allow for determination of a connecting and accepting node, the endpoint network address + port pairs are compared, and the lexically smallest one is designated 'client'.

The client sends the first authorization message.

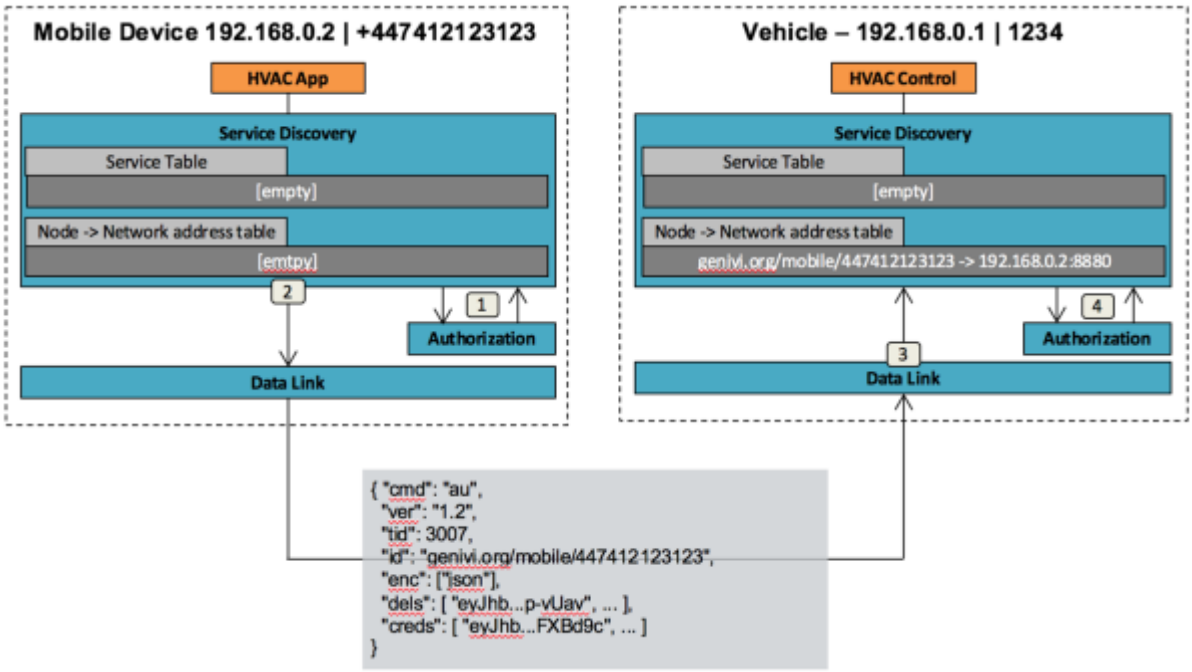

Figure 8-5 – Mobile-to-vehicle authorization

The client sends the first authorization message (see example, Figure 8-5) with the following information:

- **Version (“ver”)**
  Indicates the RVI protocol version. If the receiving side supports the offered protocol version, it must respond with a compatible version (ideally the same one), and adhere to the corresponding protocol. If it cannot use the offered version, it must disconnect. The format of the version indicator is “X.Y”, where X is the major version and Y the minor version. Differences in minor version number indicate compatibility in essentials, allowing for differences in optional features.

- **Transaction id (“tid”)**
  A message counter which increases for each message sent.

- **Node ID (“id”)**
  The node service prefix of the sending node, on the form <domain>/<type>/<uuid>. In the example given, the phone number is used as a ‘UUID’ component. The domain owner is responsible for ensuring that whatever is used is unique for the given domain and type. Note also that “+” is not a valid character for use in service names.

- **Encoding ("enc")**
  A list (array in JSON) of preferred encodings. Every node must support at least "json", but should include all encodings that it supports, in order of preference. Encodings currently supported by RVI Core are "json" and "msgp" (for messagepack). The highest-ranking encoding in the client's list which is also in the server's list will be the chosen encoding. Upon receipt of the client's "au" message, the server must send its "au" message in the current encoding, then switch to the newly chosen encoding. The client receives the server's "au" message using the current encoding, then switches to the newly chosen encoding.

- **Delegates ("dels")**
  A list of credentials used as delegates when authorizing and signing some of the credentials included in "creds". A credential will be considered invalid if it either hasn't been signed by a trusted Provisioning Server, or has been signed using a valid delegate. To be safe, any credential signed by a delegate should always be sent along with the delegate needed to validate it. Do not assume that the receiving node has cached the delegate from previous interactions.

- **Credentials ("creds")**
  A list of credentials representing the access privileges of the sending node. Each credential is encoded as a JSON Web Token (JWT).

The steps of authorization process are as follow:

8.5.1. Client requests credentials (and potentially delegates) from local Authorization

These are included in the "au" message generated.

8.5.2. Client transmits authorization package to vehicle node

The "au" message is always JSON-encoded. If another encoding is negotiated, this is activated after the "au" message.

8.5.3. Vehicle node asks Authorization to validate incoming authorization package

Validation includes

- checking protocol version
- checking supported encodings
- validating and storing each credential

Once completed, the service name prefix(es) are added together with the network address in the address table of Service Discovery.

The Vehicle device sends a similar authorization package to the mobile device, as is shown below.

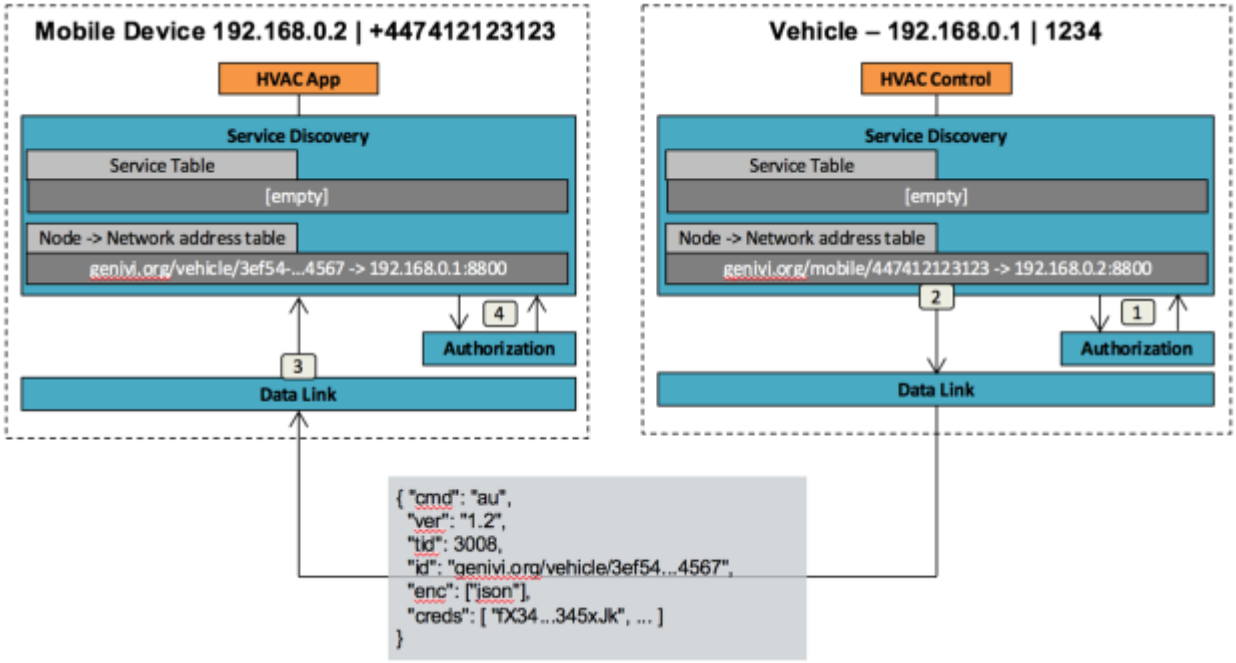

Figure 8-6 – Vehicle-to-mobile authorization

Once authorization packages have been exchanged between two nodes, they know their counterpart's address, and which service requests that should be forwarded to it.

The prefix element will be matched against the "right_to_receive" element of the credential to ensure that a node does not try to serve traffic it is not authorized to handle.

The authorization is valid on the receiving node until credentials expire or are revoked (in later change messages).

8.6. Service Announcement

When a node has sent its authorization message to another node, the receiving node uses the sending node's credentials to determine which services the receiving node should announce to the sender. By pattern-matching all available services against a received credential's "right_to_invoke" element, it can filter out those services eligible for announcement. Each node must also ensure that services to be announced match the own node's "right_to_receive" patterns, so that no services are announced that the announcing node is not authorized to handle. Pattern matches are handled according to Chapter 7.4.

A node can relay service announcements to neighboring nodes authorized to invoke those services. In other words, the same filtering logic is used when deciding whether to relay a service availability status, as when deciding to advertise a local service to a neighboring node.

In order to avoid service announcement loops, the relaying node prepends its own identity to a "route" list. The receiving node checks the "route" list for its own identity; if it finds itself in the list, it must ignore the message. A maximum length of the "route" list can be specified using a hop count limit ("hops".)

The announcement is carried out as follows:

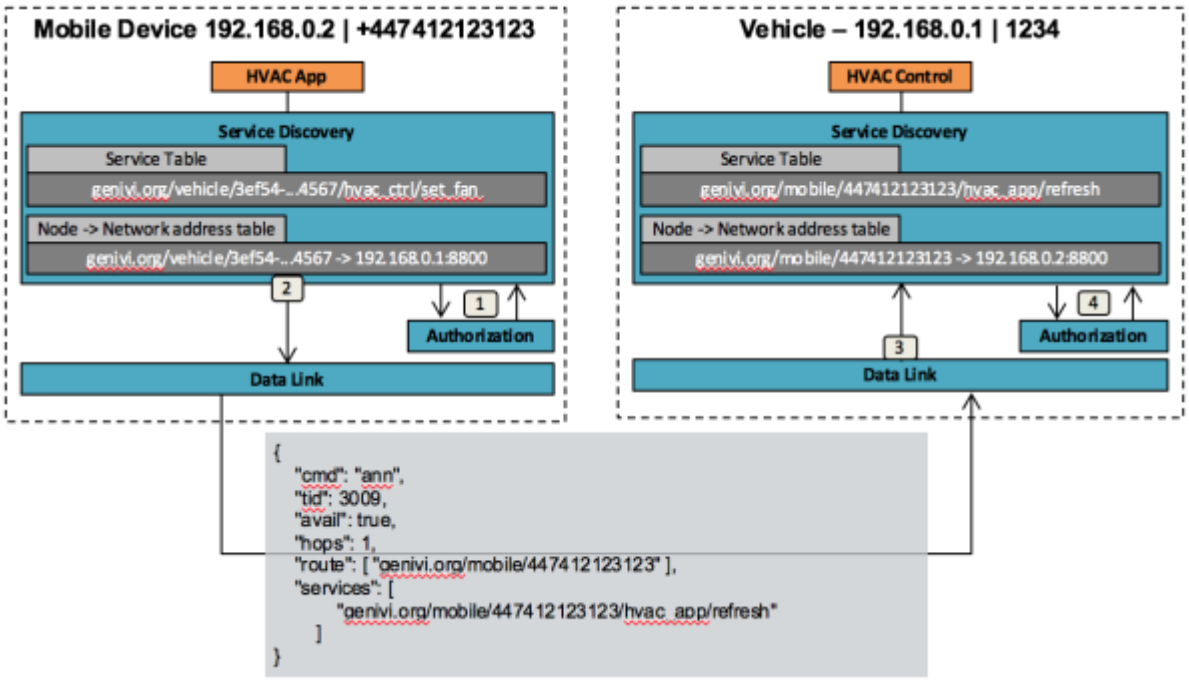

Figure 8-7 – Mobile Device service announcement

The “service announce” (“sa”) message contains the following elements:

8.6.1. Transaction id (“tid”)

A message counter which increases for each message sent.

8.6.2. Availability status (“avail”)

A boolean indicating whether the listed services are available (true) or unavailable (false)

8.6.3. Hop count limit (“hops”)

one or more services that the receiving node can invoke on the sending node. The value must be a non-negative integer, with the following interpretation

- 1 is the default, and means at most one hop – i.e. the announcement goes to the immediate neighbors only, and must not be relayed further.
- N, for any value > 1, sets a maximum length of the “route” list.
- 0 means “no limit”

8.6.4. Route list (“route”)

A list (array) of node identifiers, in reverse order, of all nodes that have sent or relayed the announcement.

The steps of announcement process are as follows:

1. **Filter available services**
   The services currently available on the sending node are filtered against the credentials of the receiving node. Specifically, only services for which the receiving node can invoke and the sending node can receive must be announced.

2. **Encode and transmit announcement message**
   The package contains all services that the Mobile can invoke on Vehicle. The “route” element is set to a list containing the sending node id.

The receiving node performs the following steps:

1. Check whether the own node Id is present in the “route” list. If so, ignore the message.
2. Check whether the “hops” value, if present and non-zero, is smaller than the length of the “route” list. If so, ignore the message.

3. Pass on service availability information to Service Discovery. Note that the “route” list needs to be kept, and linked to the current connection; if the connection goes down, all services announced via that link must be removed. There may be multiple different routes to the same remote service name.
4. If the “hops” value is greater than the length of the “route” list, proceed to check whether any services should be relayed to neighboring nodes:
    a. Filter the “services” list against the own node’s “right_to_receive” patterns. Only services for which the node can receive invocations can be relayed.
    b. For each connected node not in the “route” list, filter the “services” list against the node’s “right_to_invoke” patterns. If this results in a non-empty list, send a relay message containing the matching service names.
    c. When sending the relayed announcement, prepend the own node Id to the “route” list, substitute the list of matching service names for the original list of services, and update the transaction Id.

Once the process has completed, all services announced by the mobile device are stored in the service table of Service Discovery on the vehicle side.

The Vehicle goes through the same process of service announcement:

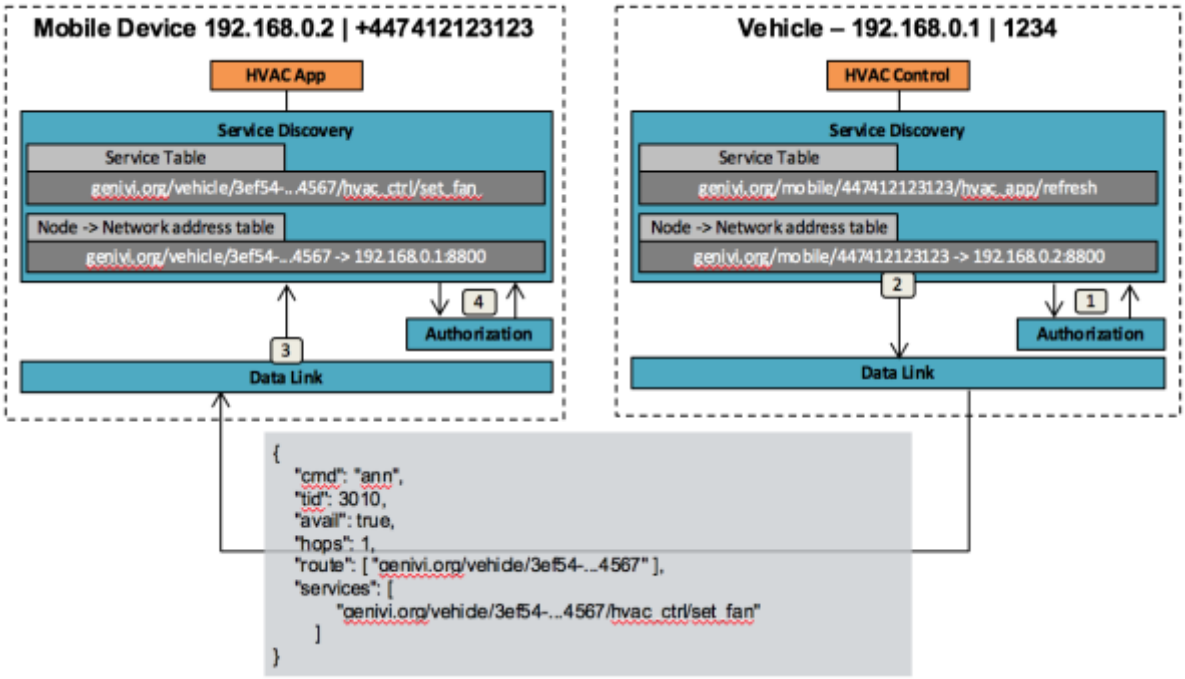

Figure 8-8 – Vehicle service announcement

New service announcements are issued if *credentials* are revoked or added, if services are registered or unregistered.

9. Request Routing

A request sent by one service to another goes through several steps. The following chapters describe a use case where a vehicle-based media player requests a song to be played from a cloud-based media server. The involved components are shown below.

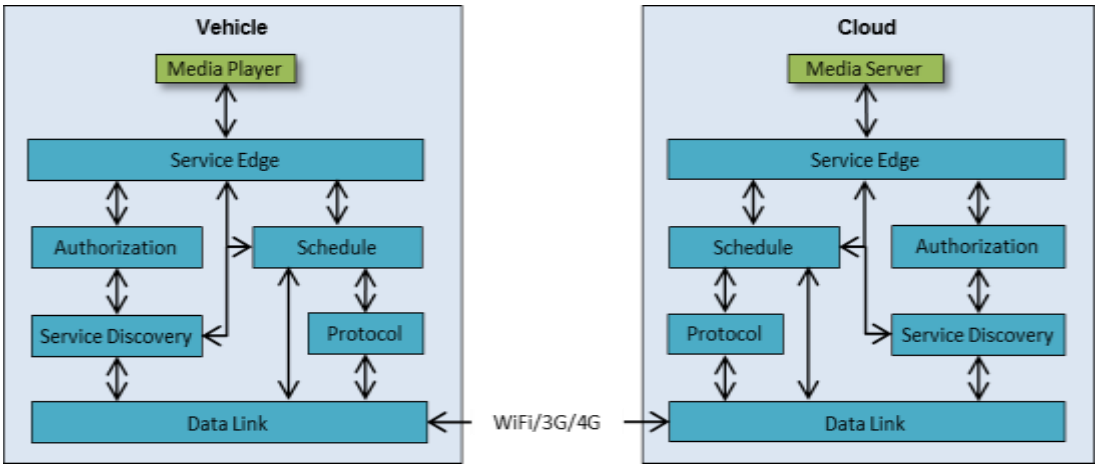

Figure 9-1 – Request Routing setup

At the start of the request routing, the following preconditions are met:

1. **Services are registered**
   The Media Player and Media Server are both registered with their Service Edge, as described in chapter 8.1.
2. **Credentials have been exchanged**
   The Vehicle and Cloud nodes have connected before and exchanged credentials.
3. **No connection**
   The mobile connection between the cloud and the vehicle is not currently active.

9.1. Step 1 [Vehicle] - Submit request to Service Edge

The transaction is initiated when the Media Player on the vehicle sends a play request, addressed to the Media Server in the Cloud, to its local Service Edge.

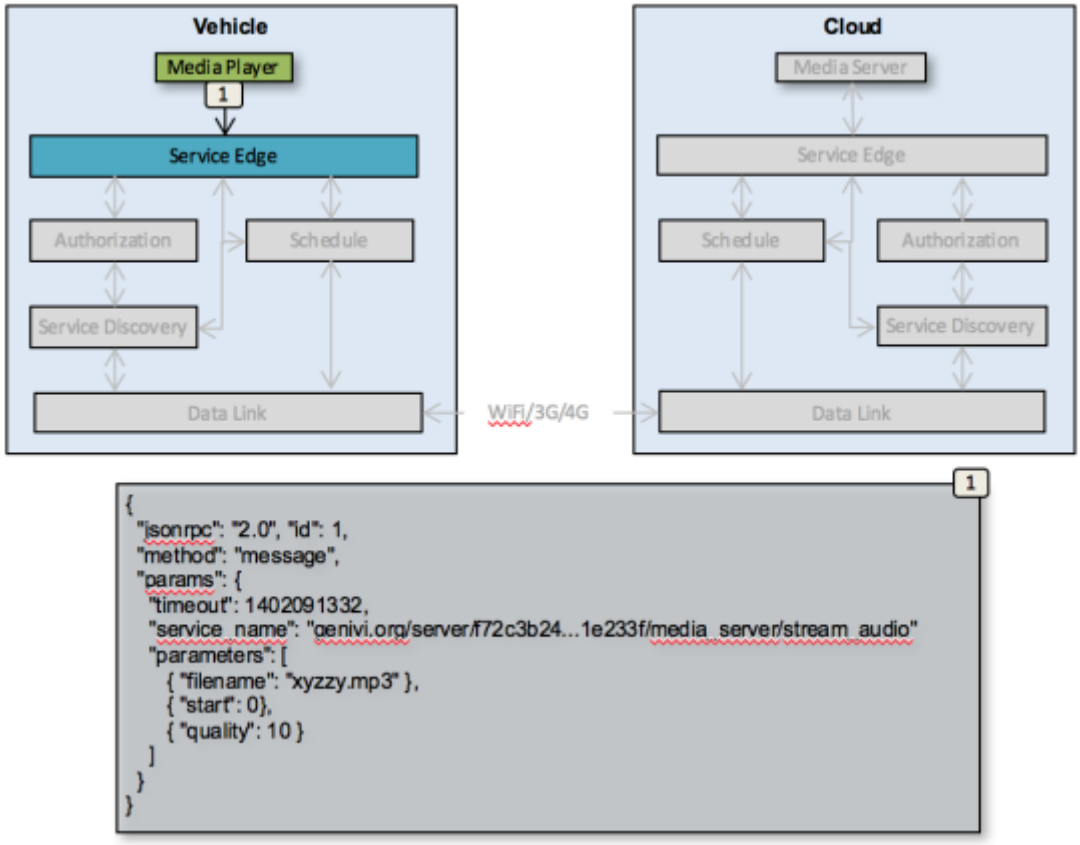

Figure 9-2 – Initial request submit

The request contains the following elements

1. **method**
   Always “message”.
2. **timeout**
   Specifies the time stamp, as a UNIX timestamp (ms), by which the reply for this request has to be returned to the Media Player in order not to trigger a timeout error.
3. **service_name**
   Specifies where the request is to be sent as a fully qualified service name. After checking the access privileges of the local node, the node Id part of the service name will be resolved to a network address by Service Discovery.
4. **parameters**
   Provides additional, arbitrary data to deliver to the target service.

9.2. Step 2-9 [Vehicle] – Validate and route message

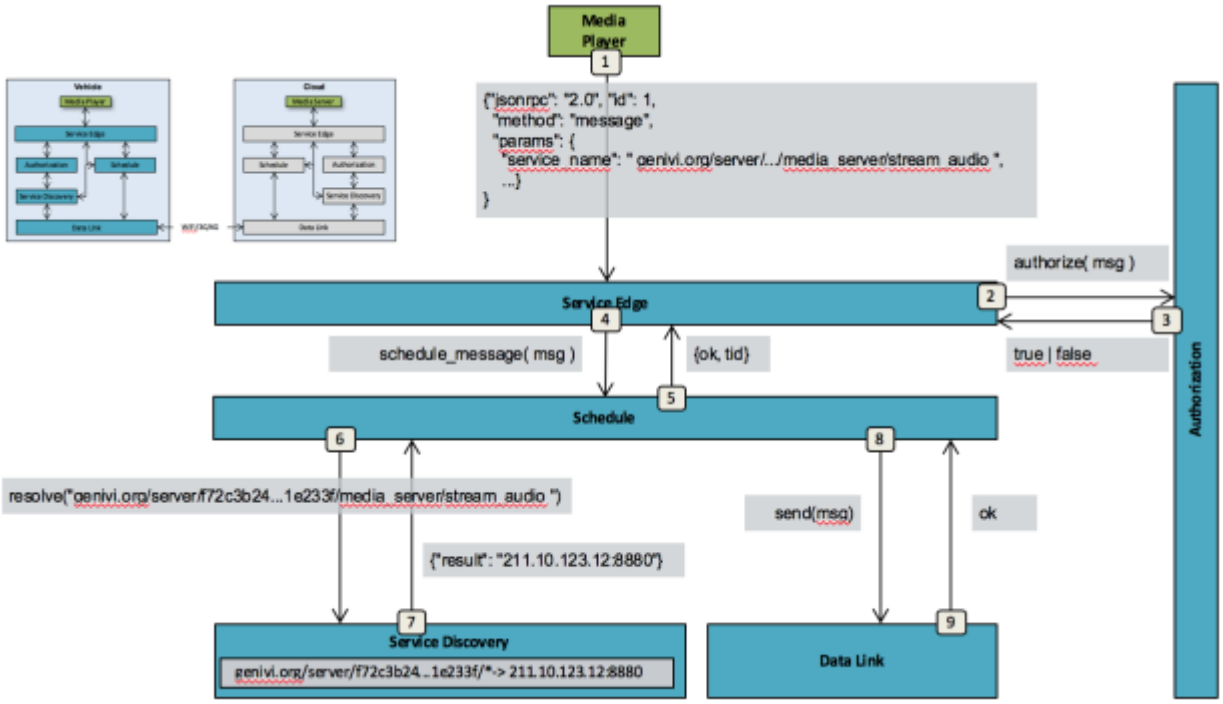

Figure 9-3 – Validation and routing

Service Edge forwards the request it received from the Media Player to Authorization to be validated.

The validation and routing process involves the following steps:

9.2.1. 2-3 – Authorize invocation

The service name is passed to Authorization for access control. In order to pass, the service name must match some pattern for “right_to_invoke” in one of the credentials for the local node. Given that the service is remote (as evident by the first three parts of the service name, or by asking Service Discovery), a matching “right_to_receive” pattern must also be found.

If the Authorization subsystem rejects the request, processing must be aborted, and an error message be returned to the client.

9.2.2. 4-5 – Schedule message for delivery

Following successful authorization, and given that the service is remote, the message, after being annotated by Service Edge, is passed to Schedule. The Schedule subsystem first generates a Transaction Id and returns to Service Edge. At this point, the Service Edge can return the Transaction Id to the client, provided that the request is not marked synchronous.

9.2.3. **6-7 – Resolve address if possible**

Schedule checks with the Service Discovery system if a connection to the target node is available. If so, the message is routed immediately. Otherwise, if the node is configured to connect on demand to the destination node, a request to connect is issued, and the message is queued for later dispatch. If no auto-connect is possible, the message is simply queued. A timer needs to be set to ensure that the message is not queued indefinitely.

Strictly speaking, what needs to be done is to check alternative routes for the given service name (a route is included in each service announcement (“sa”) message.) Any route where the next hop is unavailable can be discarded. Among the remaining routes, the “best” route is chosen. Unless some more sophisticated notion of “best” exists, any one of the shortest routes is the recommended choice.

9.2.4. **8-9 – Send to remote node**

Once a connection to the remote node has been found, or becomes active, after successful authorization handshake, queued messages are dispatched, in order, to Data Link for transmission to the remote node. The Data Link subsystem takes care of encoding the message as a “rcv” RVI message, and possibly using the fragmentation protocol if the encoded message is larger than the maximum packet size.

9.3. Step 10-14 [Server] – Remote-node processing

When the message arrives at the remote node, the following steps complete the invocation. For an asynchronous service invocation, any error in processing (e.g. a failed authorization) will lead to the message being silently dropped.

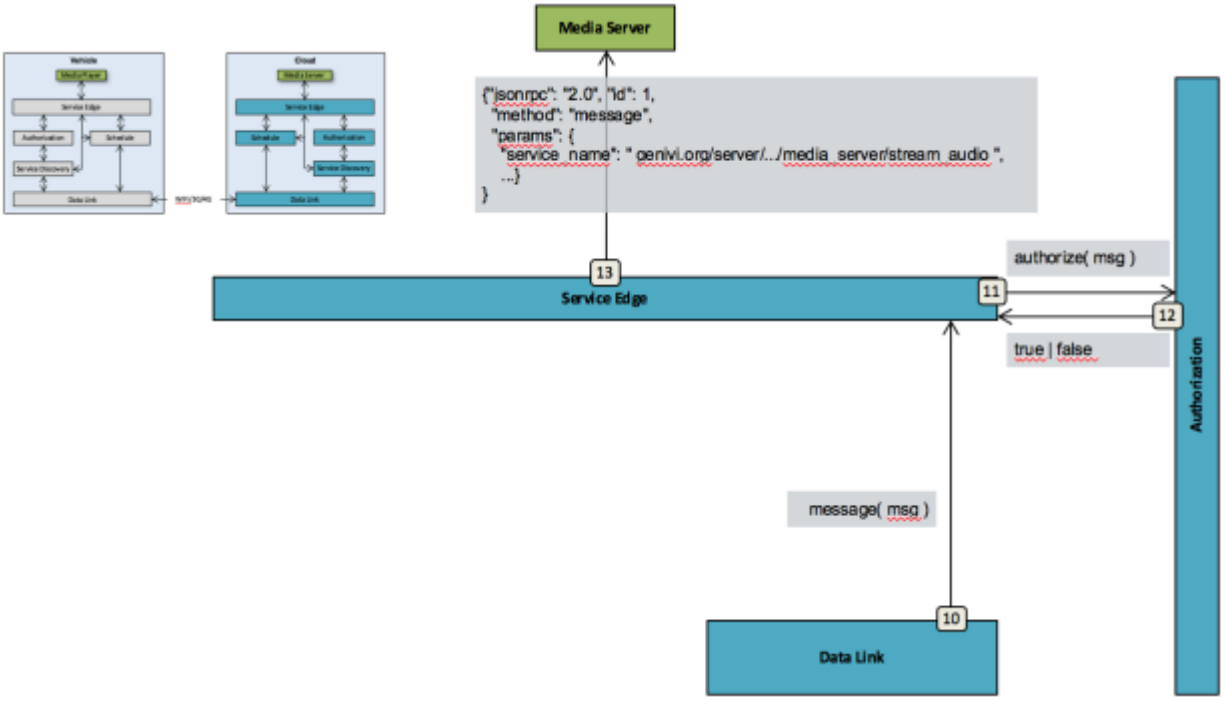

Figure 9-4 – Server-side processing

9.3.1. 10 – Data Link receives message

The Data Link layer receives and decodes the message, possibly via the fragmentation subsystem. Once the message is recognized as a "rcv" message, i.e. a remote service invocation, it hands it off to the Service Edge.

9.3.2. 11-12 – Authorization

The Service Edge first checks whether the service is available (not illustrated), then asks Authorization to check access rights, finding the originating node's credentials e.g. via the active connection. In order to pass, the service name must match a "right_to_invoke" pattern for the sending node and a "right_to_receive" pattern for the receiving node.

9.3.3. 13 – Dispatch to service

Once authorized, the Service Edge dispatches the message to the service. For an asynchronous service invocation, any return value from the service is ignored.

9.4. Invocation relay

In case there is a relay node between the sending and the receiving node, the processing flow in the relay node will look like Figure 9-5.

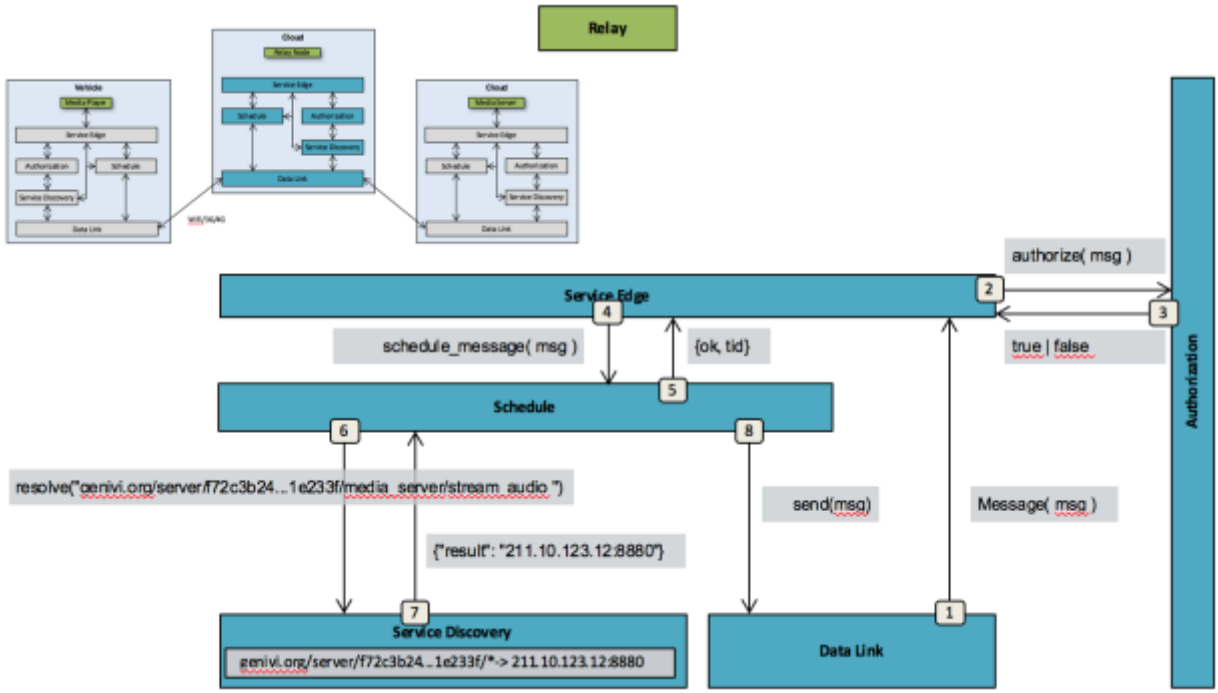

Figure 9-5 – Service invocation processing in relay node

9.4.1. 1 – Data Link receives message

Identical to chapter 9.3.1

9.4.2. 2-3 – Authorize

Identical to chapter 9.3.2, except Service Edge determines that the service name represents a remote service.

9.4.3. 4-5 – Schedule message for delivery

Identical to chapter 9.2.2

9.4.4. 6-7 – Resolve address if possible

Identical to chapter 9.2.3

9.4.5. 8 – Send to remote node

Identical to chapter 9.2.4

9.4.6. Security considerations

The node receiving a relayed request must implicitly trust that the relay nodes have not tampered, or even forged, the message. This may be a consideration leading to a decision to limit announcement- and invocation relay to well-known and implicitly trusted nodes.

9.5. Synchronous service invocation

A client can instruct the Service Edge to wait for a reply from the called service to be delivered before returning, making the invocation a proper Remote Procedure Call (RPC). This is done by adding “synch”: true to the “message” request. This affects processing in the following ways, at a minimum:

1. The Service Edge on the sending node performs Authorization validation as usual, then creates an Internal Service Point (ISP, see below) and adds an element, “reply_id”: <ISP>, to the message.
2. The Service Edge on the receiving node notes the “reply” and “reply_id” values, captures the reply from the service dispatch, and packs it into a “rcv” RVI message addressed to the ISP.
3. A “rcv” message addressed to an ISP bypasses Authorization on all nodes.
4. The Schedule subsystem must recognize internal service names, peel off the ‘$’ and route on the node Id.
5. The ISP, when receiving the reply, needs to format and deliver a reply message to the client.
6. If no matching ISP exists on the node matching the ISP’s node Id, the message must be discarded.
7. At a minimum, if the processing flow fails, the ISP must respect the “timeout” value of the request and deliver a timeout message.
8. Each RVI node should note the “reply” status and deliver an error reply as soon as possible if processing is aborted.

9.5.1. Internal Service Point

An Internal Service Point needs to be a unique internal service name for security purposes. This is achieved by prepending a ‘$’ to the node Id, and adding an ephemeral service name part (e.g. a unique serial number). See example in Figure 9-6.

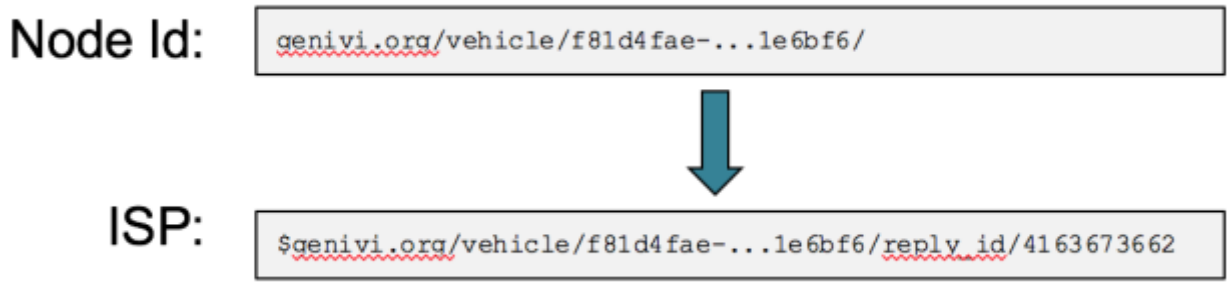

Figure 9-6 – Generating an Internal Service Point

9.5.2. Security considerations

When sending or receiving a “rcv” message addressed to an ISP, as noted in item 3 above, no authorization check is performed. While this means that security is somewhat weakened, the Service Edge is responsible for rejecting any messages from outside, addressed to RVI service names, and ISPs are ephemeral and created on demand.

9.6. Handling attachments

When sending large data objects, protocols like JSON-RPC over HTTP are often insufficient. This chapter will not deal with how to send or receive attachments via the Service Edge, but rather specify how they are represtented within RVI.

The “rcv” message supports a “files” element, where the value is a list of objects on the form:
{ “cid”: <content id>,
“hdrs”: [<http header>],
“data”: <data> }

where <http header> is an object, {K: V}, i.e. the HTTP header encoded as a JSON object.
The payload of a given attachment can be referenced from within the request parameters as “file:<content id>”. The “hdrs” element is optional – depending on how the request entered the system, now headers may have existed in the first place. If the request arrived via HTTP (e.g. using a combination of JSON-RPC and multipart POST, the headers should be preserved, and reused if the final service dispatch uses HTTP as well. If the service does not use HTTP, the “files” element should be included as-is.

10. Node and Service Provisioning

Before a Node, and its locally connected services, can access remote services on other Nodes, it needs to be provisioned with a certificate specifying its access rights, public keys, and other information. See chapters 0 and 8.2 for details.

All certificates in an RVI system are signed by the private key of a provisioning server. The public counterpart of that key is installed in all nodes' Authorization as a part of the RVI software install and setup process.

10.1. Provisioning flow

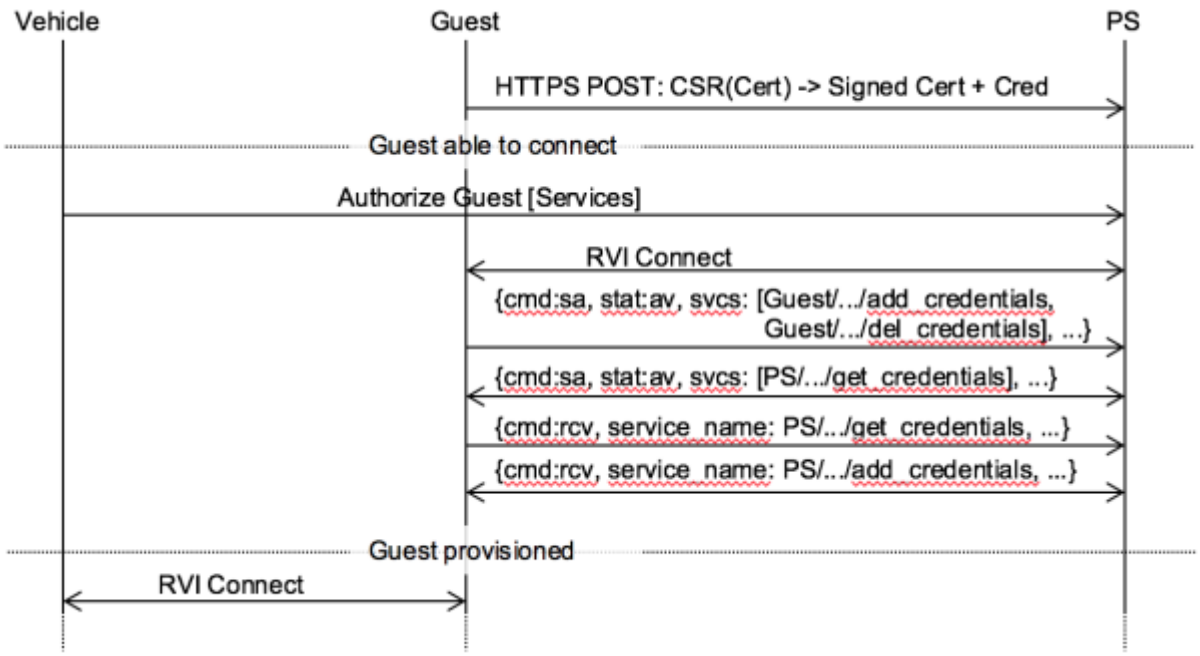

Figure 10-1 – Provisioning flow

Figure 10-1 illustrates a skeleton provisioning flow:

1. Guest creates an unsigned X.509 certificate
2. Guest sends a Certificate Signing Request (CSR) to the Provisioning Server (PS), passing the unsigned cert for signing. PS returns the cert, signed with its private key, and also creates and signs a credential file authorizing Guest to fetch further credentials from PS via the RVI protocol.
3. Guest is now able to connect to any device that recognizes PS as Root CA, but is not yet authorized to do anything except fetch more credentials.

4. Vehicle (owner) authorizes Guest to access certain services. (if supported, and so configured, PS could in the future push credentials to Guest)
5. Guest connects to PS using RVI, authenticating itself with the newly signed certificate
6. Since PS is authorized to use the Guest/creds service, Guest announces its availability.
7. Since Guest has been authorized to use the PS/get_credentials service, PS announces its availability.
8. Guest calls PS/get_credentials, asking for new credentials.
9. PS calls Guest/add_credentialss, passing found credentials for Guest.
10. Guest can now connect to Vehicle and use the authorized services.
11. Guest connects to vehicle ...

Note that step 2 above, where the Provisioning Server responds to the Certificate Signing Request, would involve authentication and authorization of Guest. The RVI architecture does not address this problem. Instead, it is the responsibility of the designer/maintainer of the Provisioning Server to choose and implement a strategy.

10.1.1. **RVI credential handling services**

The provisioning flow above assumes the presence of internal RVI services that respond to requests for credentials (Provisioning Server side) and receive new credentials and credential revocation instructions (client side). The services reside under the service name tree
<domain>/<type>/<uuid>/RVI/provisioning/

Note that initial credentials must be installed that allow access to these services. For the Guest node, these credentials are provided in step 2 above.

10.1.2. **get_credentials service**

This service runs on the Provisioning Server. Example request:

```
{
  "jsonrpc": "2.0", "id": 1,
  "method": "message",
  "params": {
    "timeout": 1402091332,
    "service_name": "genivi.org/server/f72c3b24...1e233f/RVI/provisioning/get_credentials"
    "parameters": []
  }
}
```

Figure 10-2 – Example credential request

The guest does not need to identify itself, since the Provisioning Server would not be able to take the guest's word for it, anyway. The identity of the guest is determined by the authorization process of the connection handshake. Security is ensured further by validating the request against the access control information in the signed credentials.

Parameters: none

The Provisioning Server must not respond if the validation process fails, but must respond with a "add_credentials" service invocation, even if it contains no credentials.

10.1.3. **add_credentials service**

This service is run on the guest/client node and is used to introduce new credentials. As above, the Provisioning Server's identity is derived from the authorization process of the connection handshake, and the right of the Provisioning Server to send us credentials is guaranteed through the credentials.

Parameters: A list of JWT-encoded credentials (see chapter 8.2)

The Provisioning Server must invoke this service in response to a valid "get_credentials" request. It should also invoke this service any time when there are new credentials to deliver.

10.1.4. **del_credentials service**

This is the corresponding request to remove existing credentials.

Parameters: A list of "jti" (JWT ID) values (see chapter 8.2), identifying the credentials to remove.

The Provisioning Server should invoke this service as soon as possible when there are credentials that need to be removed.

10.2. P2P Access Granting

If a node wants to grant another specific node access to its services without having to invoke a remote provisioning server, the granting node (which will give out access rights) can issue a P2P certificate with specified access rights to the given services. This certificate follows the same specification and use cases as regular certificates, with the single exception that it is signed by a private key owned by the granting node.

Once created, the granting node distributes the certificate to the granted node (which will receive access rights) as described in chapter "" and subsequent chapters.

The granted node can then use the certificate to sign its requests to the granting node, which will use the public key of the certificate to validate the request.

The created P2P certificate is only useable between the two given nodes since it explicitly gives access to the granted node (through the certificate's "sources" element), and has been signed by the granting node. The granted node cannot use the certificate anywhere else since no other node has the certificate's public key to validate the request.

11. Service Edge feature set

Service Edge supports a number of operations toward connected services, Authorization, and Service Discovery.

11.1. Local Service Registration

A local service will be able to connect to Service Edge and register itself as described in chapter "". Once successfully registered the local service shall be able to send and receive requests, as well as subscribe to and receive service availability reports from Service Edge.

11.2. Service availability reporting

When a remote or local service becomes available, other services, as well as Service Discovery, shall have the option of being informed of the event.

A subscription command sent to Service Edge will contain a service name prefix for which any matching services shall be reported.

There are two events that can trigger a service availability report to a locally connected service or Service Discovery.

First, a service availability report may be sent by Service Discovery as a reaction to a received service announcement from Data Link, which means that there is now a data link available to a remote service that matches the subscribed-to service name prefix. In these cases a locally connected service will be informed that a remote service is available.

Second, a local service may register directly with Service Edge, thus making the service available for locally and remotely originated requests. In these cases, Service Edge will report the event to Service Discovery, a service availability subscriber who in its turn will forward it to relevant remote nodes.

11.3. Process requests from local services

A locally connected service may send requests (RPCs, replies, messages, and metrics) to Service Edge for processing and forwarding to its target service. See chapter “” for details.

11.4. Process requests from remote services

Service Edge can also receive incoming requests from Protocol, which are to be forwarded to a locally connected service. Service Edge will use Service Discovery to map

12. Service Discovery feature set

12.1. Register local services

Service Edge shall register locally connected services with Service Discovery, specifying their network address and their service names to match incoming requests against. The received information will be used to resolve the targeted service name to the network address of the locally connected service that can handle the request.

12.2. Register node to network address mappings

Service Discovery will also receive information about remotely available services from Data Link. Such services will be stored, with their corresponding network address, in Service Discovery. When the communication channel supporting the remote service communication disappears, Service Discovery will be informed.

Any subscribers to remote service availability will be informed of remote services becoming available and unavailable, as long as the service name matches the pattern provided with the subscription request.

12.3. Resolve service to network address

Service Discovery will be able to map a name, describing a service, to a network address that requests can be sent to. Information about the service name-to-network address mapping is provided by the two authorize and announce stages described in chapter “”.

12.4. Process incoming service announcements

Data Link will, during its authorize and announce stage, receive service announcements from other nodes with information about network addresses, and services. The remote service information is

forwarded by Data Link to Service Discovery, which will match the reported services against its subscription tables and send out reports. Service Edge is a typical such subscriber and will, in its turn, forward the information to its locally connected services.

12.5. Send outgoing service announcement

When Data Link reports that a remote node is available for communication through its authorization step, Service Discovery will be informed.

Service Discovery will respond by filtering all available locally connected services through the "destinations" element of the certificate provided by the remote node. See chapter "**Error! Reference source not found.**" for details.

All local services matching the destinations entry will be forwarded to Data Link, which in its turn will construct an announce message with the service list to the remote node. See chapter "" for details.

12.6. Process communication channel availability reports

Data Link will report when communication channels to other nodes become available and disappear. Such reports will trigger service availability reports to Service Edge, who will forward them to their subscribing services.

13. Authorization feature set

13.1. Authorize local requests

Once registered, locally connected services will submit "invoke" requests to be sent to a targeted remote service. See chapter "" for details.

Service Edge will forward the received "invoke" request to Authorization to have it validated. Authorization will check that the signature, generated by the service's private key, can be verified with its public key.

Authorization will also verify that the Node that both Service and Authorization executes on has the right to invoke the targeted service. This is done by pattern matching the "destinations" element of the Node's certificate against the name of the targeted service. See chapter "**Error! Reference source not found.**" for details.

13.2. Authorize remote requests

A remote request received and decoded by Protocol and forwarded to Service Edge, will be sent to Authorization for validation. See chapter “” for details.

The request will contain a certificate, describing the rights of the sending node, and a signature generated by the private counterpart of the certificate’s public key. Authorization will validate the certificate, signature, and the remote node’s access rights to the targeted service.

13.3. Provision certificates

Authorization must provision, delete, and blacklist certificates when directed to do so from a provisioning server. Please see chapter “Node and Service Provisioning” for details.

14. Schedule feature set

14.1. Process local requests

A request received from local Service Edge will be stored until a data link to its target node is available. If the request times out before such a link becomes available, or it times out before a reply has been received (when applicable), a timeout is sent back to Service Edge to be forwarded to the service originating the request.

14.2. Process remote requests

Remote requests received from Protocol needs to be stored in cases where the targeted locally connected service is not available. Timeouts have to be sent back to the originating Node in case the service does not become available to process the request within the timeout interval.

14.3. Process data link availability reports

When Data Link reports that a communication channel, and its associated services, is available, Schedule will traverse all pending requests received from locally connected services. Those requests targeting now available services are forwarded to Protocol for encoding and transmission to the remote Node.

14.4. Process service availability reports

When Service Edge reports to Schedule that a locally connected service is available, Schedule will traverse all pending requests received from remote nodes. Those requests targeting the now available local service are sent to Service Edge to be forwarded to the service itself.

15. Data Link feature set

15.1. Setup communication channel

Data Link shall be able to setup a communication channel to a remote node, identified by a network address. Once the communication channel is up, a report is sent to any subscribing local processes, such as Schedule.

15.2. Disconnect communication channel

When a communication channel has been idle for a period of time, or is explicitly ordered to be disconnected, Data Link shall shut it down and free the associated resources.

15.3. Transmit data payload

In cases where Protocol, instead of transmitting a data packet itself, elects to have Data Link transmit the payload, Data Link shall forward the payload to its remote counterpart.

15.4. Receive data payload

When Data Link receives a data payload from a remote counterpart, it shall forward it to a suitable Protocol for further processing.

15.5. Send node authorization

When Data Link has established a communication channel with a remote node, it shall send an authorization package as described in chapter “”.

15.6. Receive remote Node authorization

When Data Link has established a communication channel with a remote node, it shall be prepared to receive and process an authorization package as described in chapter “”.

15.7. Send service announcement

Once Data Link has authorized a remote counterpart, it shall send a service announcement package to the remote Node as described in chapter “”.

15.8. Receive service announcement

Once Data Link has sent its own authorization to a remote counterpart, it shall be prepared to receive and process a service announcement from the remote node as described in chapter “”.

15.9. Report communication channel availability

Data Link shall report when communication channels to remote nodes becomes available, or disappear, to all local components who have subscribed to such information. A typical example of such a subscriber is Schedule.

15.10. Fragmentation

The data link layer supports transparent fragmentation of large packets. Unlike ordinary RVI messages, the fragmentation protocol includes acknowledgment and retransmission as-needed. It can also be used to send a single message (single ‘fragment’) with acknowledgement. See chapter 16 for details.

16. Fragmentation

RVI is designed to be agnostic in regard to connectivity options and intermittent connectivity. One consequence of this is that large messages may have to be partially transmitted via one type of connection, and completed on another. The fragmentation protocol described below allows for varying Message Transfer Unit (MTU) and lets the remote client request fragments as needed.

The fragmentation protocol is based on the receiving side requesting fragments as-needed, and re-assembling the message using the fragments received.

Full support of message fragmentation is not mandatory, but at least a part must be recognized. Specifically

16.1. Operating principle

The fragmentation support is intended to operate immediately on top of the transport layer. In essence, the sending side (Vehicle) asks the fragmentation support to deliver a message. The fragmentation support determines whether fragmentation is needed. If it is, it will create a first fragment, encode it and send it to the receiving end (Cloud).

Conceptually, the fragmentation support can be viewed as an internal server, which holds the whole message in a buffer, assigning it a unique Id and serving the receiving side with fragments. On the receiving side, a corresponding server re-assembles the message, identifying possible holes (e.g. the result of a failed fragment delivery) and issuing new requests for fragments. When a complete message has been assembled, the receiving side sends an acknowledgement message ("frg-end".)

The fragmentation support can operate over a transport using its own fragment/reassembly method (such as TCP), but does not require it, or makes any such assumptions.

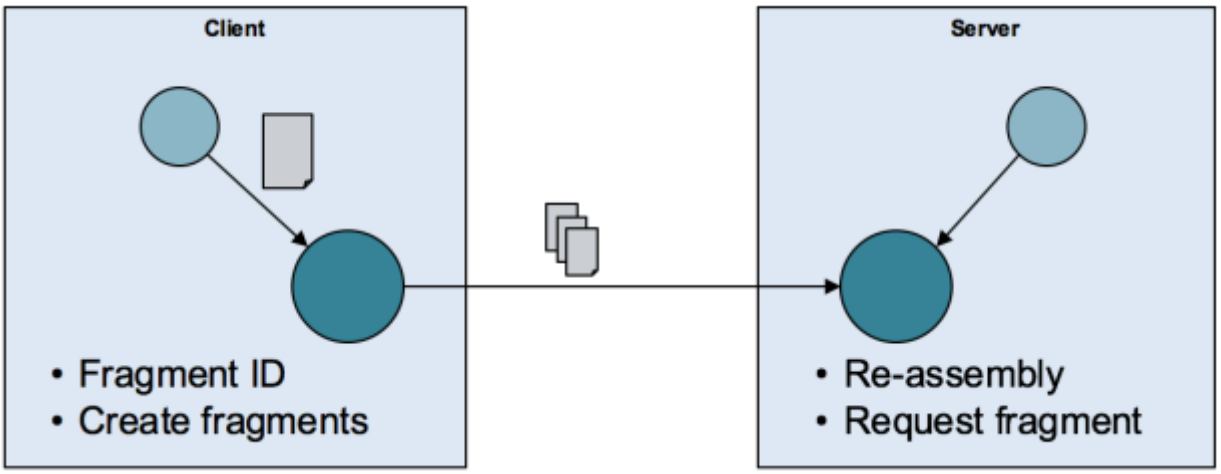

Figure 16-1 – Fragmentation operating principle

The fragmentation protocol does not specify any particular encoding method. In this document, JSON notation is used. In practice, a byte-oriented JSON-like encoding, like msgpack would be more suitable.

16.2. Messages

Fragment message:
{ "frg": [id, size, offset, fragment] }

The sending side initiates fragment transfer by sending a first fragment. The size of the first fragment is determined by the sending side. Subsequent fragments are requested by the receiver, in which case the receiver also decides how large a fragment it wants. The sender is allowed to send a smaller fragment, but not a larger one.

Note that size denotes the size of the *whole* message, not the fragment.

Fragment request message:
{ "frg-get": [id, offset, size] }

This message is sent by the receiving side in order to request the next fragment. The offset will typically be the position following the most recently received fragment, but could also represent a "hole" in the message from a missing fragment.

Fragment acknowledgement message:
{ "frg-end": [id, result_code] }

Fragment error message:
{ "frg-err": [id, result_code] }

- id (string): Message identity. This value needs to be unique within the scope of the current connection.
- size (integer): A positive integer denoting either the size of the whole message (as in the "frg" message) or the size of the requested fragment (as in the "frg-get" message).
- offset (integer): A positive integer denoting the starting byte of the fragment, relative to the whole message. The first fragment starts at 1.
- fragment (binary): A byte sequence denoting the current fragment. Note that the "frg" message doesn't contain a size indicator for the fragment. However an encoding such as msgpack does include a size indicator.
- result_code (integer): A number denoting the outcome of the transfer and reassembly. A zero (0) means all went well; a negative number indicates failure. Predefined values are:

Code or range	Definition
0	Message was successfully transferred and reassembled
-99 ... -1	Reserved for standard error codes
-1	Unknown message (i.e. in a "frg-err" response to a "frg-get" message)
-2	Protocol error
-3	Timeout error
-4	Fragmentation not supported
< -99	Application-defined error codes

16.3. Example

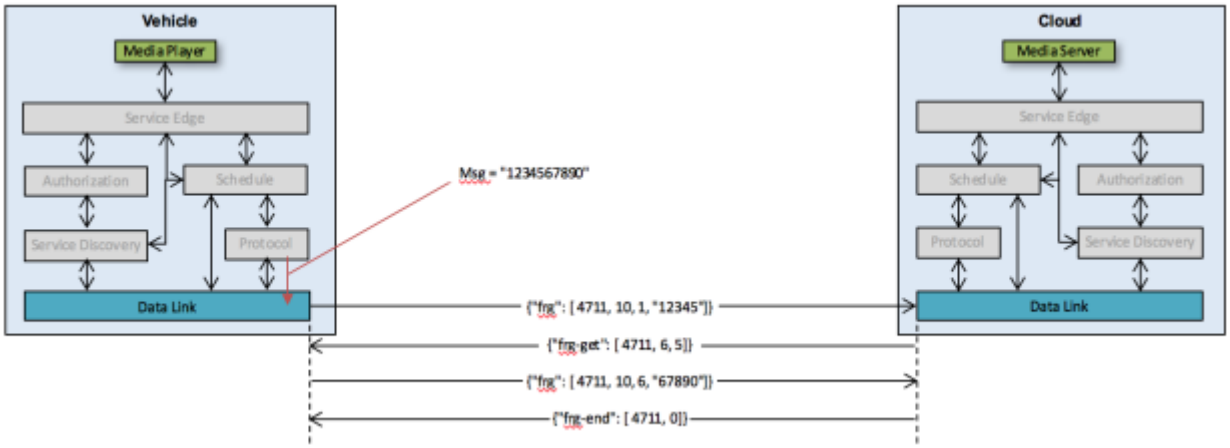

Figure 16-2 – Fragmentation example

16.4. Minimal support

Support of the full fragmentation protocol is not mandatory, but for minimal support, at least a {“frg”, [id, offset, msg]} message must be recognized. If it contains a complete message, the receiving side should respond with {“frg-end”, [id, 0]} (successful transfer); otherwise, return a {“frg-err”, [id, -4]} (fragmentation not supported).