Authorization System SysDD G3.2

**Abstract**

This document provides an overview on the Authorization System within Arrowhead generation 3.2.

This System is primarily implemented based on X.509 certificates.

1. System Design Description Overview

Table 1 System Information

|  |  |
| --- | --- |
| **Full Name** | Authorization System G3.2 M3 |
| **Owner** | Csaba Hegedűs, AITIA Inc., hegeduscs@aitia.ai |

This System has:

* A database that describes which Application System can consume what Services from which Application Systems (intra-Cloud access rules)
* A database that describes which other Local Clouds are allowed to consume what Services from this Cloud (inter-Cloud authorization rules)
* A storage of X.509 certificate PublicKeys for every Application System in the Cloud

The purpose of this System is therefore to:

* Provide AuthorizationControl Service (both intra- and inter-Cloud)
* Provide a TokenGeneration Service for allowing session control within the Local Cloud

The purpose of the TokenGeneration functionality is to create session control functionality through the Core Sytems. The output is an ArrowheadToken that validates the Service Consumer system when it will try to access the Service from another Application System (Service Provider). This Token shall be primarily generated during the orchestration process and only released to the Service Consumer when all affected Core Systems are notified and agreed to the to-be-established Service connection.

This System (in line with all core Systems) utilizes the X.509 certificate Common Name naming convention in order to work. This means that the CN is structured as it is described in the generic G3.2 M3 System-of-System Design Document (SoSDD).

1. Services and Use-cases

This System only provides two Core Services:

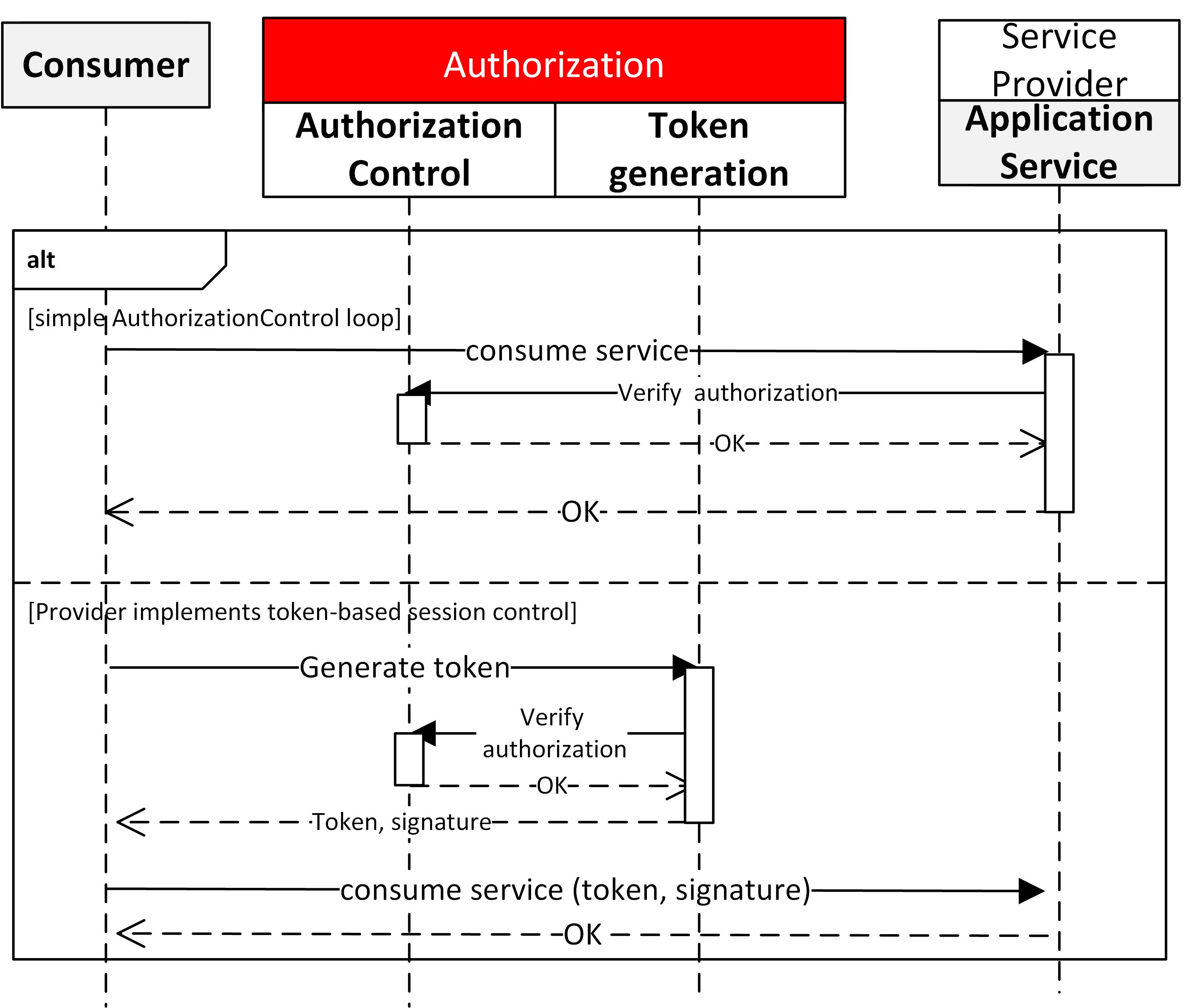
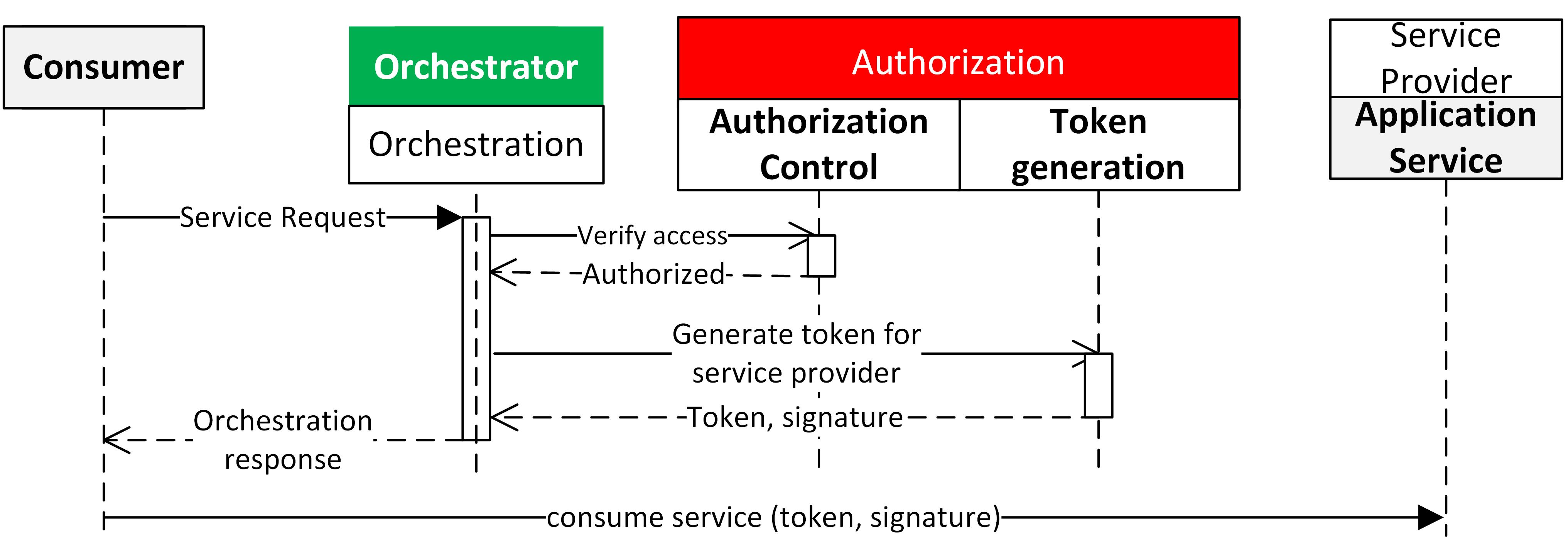
* AuthorizationControl
* TokenGeneration

There are two use cases connected to the Authorization System:

* Check access rights (invoke the AuthorizationControl)
* Generate an access token (the Orchestrator invokes the TokenGeneration)

These use cases are depicted on Fig. 1.

**Figure 1. Authorization crosscheck during orchestration process**

****The direct access of the Authorization Services is not recommended, but allowed, when the “enable\_auth\_for\_cloud” flag is set true in the app.properties. Fig. 2. depicts how that can be utilized when the orchestration process is not connected to the authorization loop.

**Figure 2. Authorization loop without ochestration**

1. Internal structure

This module is a Java jar executable. It uses the config folder contents, where the configuration files are. The code includes the following classes:

* **Main class**: instantiates the HTTP and/or HTTPS servers based on the properties files and command line arguments
* **AuthorizationResource**: contains the REST interface functions
* **TokenGenerationService**: contains the “business logic” behind the REST interface (related to the X.509 certificate handling and encryption)
* **AccessControlFilter**: implements the CN-based filtering on who can access the Authorization Services.

Used libraries:

* Java Jersey API
* Grizzly servlet container
* Hibernate ORM or DNS-SD Java client library

# Usage

The app.properties of the System includes and extra Boolean field: “enable\_auth\_for\_cloud”. If this is set to “false” and the System is started in secure mode (with the command argument “-m secure” or “-m both”), only the Orchestrator and Gatekeeper Systems are authorized to consume the AuthorizationControl or TokenGeneration Services.

The use of this module assumes that all Application Systems that are registered in the database has its X.509 certificate public key serialized in Base64 encoding stored in the database, in the “authentication\_info” field. This enables for the token generation.

Start the module as a Java executable. The following command line arguments are available:

* “-m <mode>” sets the server mode
  + “secure”: using HTTPS
  + “insecure”: using simple HTTP (default mode)
  + or “both” when two ports are used, one for simple HTTP and one for HTTPS server
* “-daemon” (Linux only!): starts the module in daemon mode, kill signals will prompt a normal shutdown, and the core system will de-register its services from the Service Registry.
* “-d”: starts the module in debug mode, which means every incoming REST request (URL + payload) and the corresponding response will be printed to the console output.

1. Revision history

# Amendments

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Date | Version | Subject of Amendments | Author |
| 1 | 2017-09-29 | M2 | Initial | Csaba Hegedűs |
| 2 | 2018-02-14 | M3 | Updating document to M3 | Csaba Hegedűs |

# Quality Assurance

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Date | Version | Approved by |
| 1 |  |  |  |
| 2 |  |  |  |