CISTER - Research Center in
Real-Time & Embedded Computing Systems

Arrowhead Programming

Michele Albano

Instituto Superior de Engenharia do Porto

INESCTEC

29-Aug-17

Outline

- Setup of Raspberry PI
- Arrowhead installing
- Arrowhead programming

Raspberry Pi

- Let us configure a Raspberry Pi
- Problem: no eth0 by default
- Solution:
  - `fdisk -l name_of_image.img`
  - Compute offset = *start* of a partition * sector size
  - `mount -o loop,offset=xxxxxx name_of_image.img /mnt/whatever`
  - Change what you want
  - `umount /mnt/whatever`
- What do I want to change?
  - /etc/network/interfaces

eth0 on by default

- Changes to /etc/network/interfaces:
  - `auto lo eth0`
  - `iface lo inet loopback`
  - `iface eth0 inet static`
  - `    address 192.168.69.10`
  - `    netmask 255.255.255.0`
  - `    gateway 192.168.69.1`
- BUT the last RPI doesn’t have eth0
  - For example, mine has enxb827eb444b14

sshd on by default

- Go to the first partition of the memory card
  - `Touch ssh`

Now you can burn the image to the memory card

- For example,
  - `dd if=name_of_image.img of=/dev/sdf`
- Then,
  - `ssh pi@192.168.17.202`
- On the Raspberry:
  - `sudo raspi-config`
  - And then switch on ssh permanently
  - And maybe enlarge the filesystem to the whole memory card (menu “advanced”)

To complete your setup

- Share the internet connection of your computer, to be able to reach internet from the raspberry pi

Outline

- Setup of Raspberry PI
- Arrowhead installing
- Arrowhead programming

- Download Arrowhead distribution
  - https://drive.google.com/file/d/0B8k8mA0XtZUdVjI5SVZISXNocmc/view?usp=sharing
  - You have actually it on the raspberry pi image that was provided
- Then, please install
  - mysql DB server, used by the core systems
  - phpMyAdmin, to manage the DB easily
  - postman, to have easy access to core services' functionalities

Core modules - overview

- **Service Registry**: Arrowhead Systems register and revoke the Services they offer. This entry also includes the http endpoint where the Service is accessible.

- **Orchestrator**: Arrowhead Systems turn to the Orchestrator with Service requests if they wish to consume Services. The initiated orchestration process returns with a single one or a list of Service Providers. After orchestration, the requester System has to consume the specified Service from the specified Provider(s).

- **Authorization System**: the Orchestrator queries this Core System for checking authorization information. This System stores the intra-cloud and inter-cloud access rights. It is also responsible for issuing authorization token – if enabled.

- **Gatekeeper:** this Core System handles all the inter-cloud orchestrational tasks among Arrowhead Local Clouds. They provide two Services for their Orchestrators: the Global Service Discovery and Inter-Cloud Negotiations services.

- **Arrowhead API**: this module is a simple REST tool to provide CRUD operations on the core database.

Setting up a database

- This framework uses MySQL server through the Java Hibernate ORM
  - https://dev.mysql.com/downloads/installer/
  - It is recommended to use a database manager GUI (e.g. the built-in MySQL Workbench)
  - Username and password is arbitrary
  - Remote access for the account is needed, if the Core Systems will run on different machines
- Please import the ***create_arrowhead_database_1.sql*** script
  - This will create the database schema „arrowhead" with all the tables and inserts dummy entries for the examples showed later in this guide
- Please create a "logs" table in the database
  - `mysql –u root –p`
  - `database create logs;`
- Please import the ***create_logs_table.sql*** script
  - This will create a „log" table by default; and will be used by the Core Systems for joint logging
  - Multiple instances of this table can be created if separate logging is expected for each Core System (configurable in each Core System separately)

Generating Certificates (optional)

If secure connections are required (using SSL), the Core Systems have to be installed with the appropriately created X.509 certificates stemming from the master Arrowhead CA. There are two options:

- Using the testcloud certificate sets (testcloud1 and testcloud2)
- Creating an own set of certificates

For the latter scenario, it is necessary to take steps that are not part of this tutorial, including

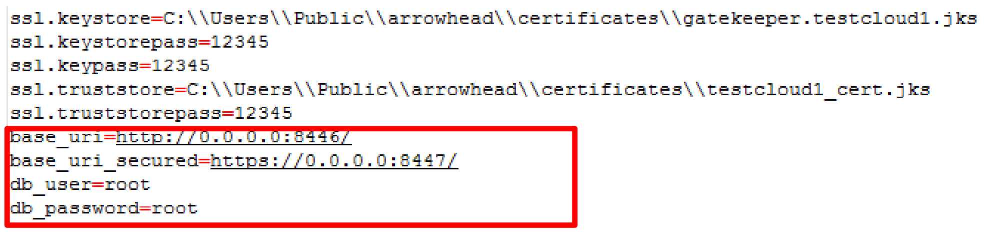
- Installing a freeware certificate manager tool such as KeyTool Explorer
  - http://www.keystore-explorer.org/ available for all platforms

Using certificates are neither mandatory nor adviced through this guide. All Core Systems can be started using unsecured http (advised).

Configuring the core modules

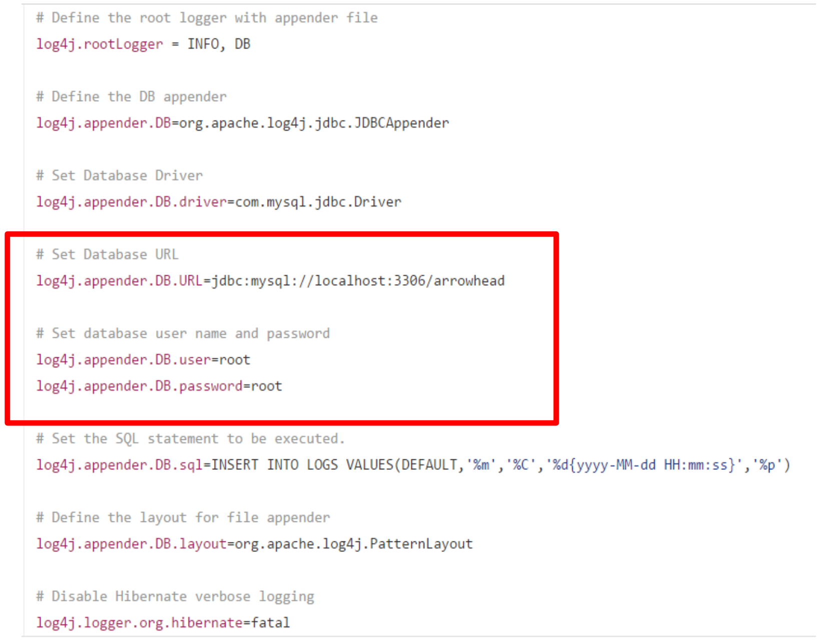
- Every module is a runnable Java-jar file and has two properties file:
  - The „config/app.properties” includes general configuration
  - The „config/log4j.properties” configures the logging.

The app.properties file:



```
ssl.keystore=C:\\Users\\Public\\arrowhead\\certificates\\gatekeeper.testcloud1.jks
ssl.keystorepass=12345
ssl.keypass=12345
ssl.truststore=C:\\Users\\Public\\arrowhead\\certificates\\testcloud1_cert.jks
ssl.truststorepass=12345
base_uri=http://0.0.0.0:8446/
base_uri_secured=https://0.0.0.0:8447/
db_user=root
db_password=root
```

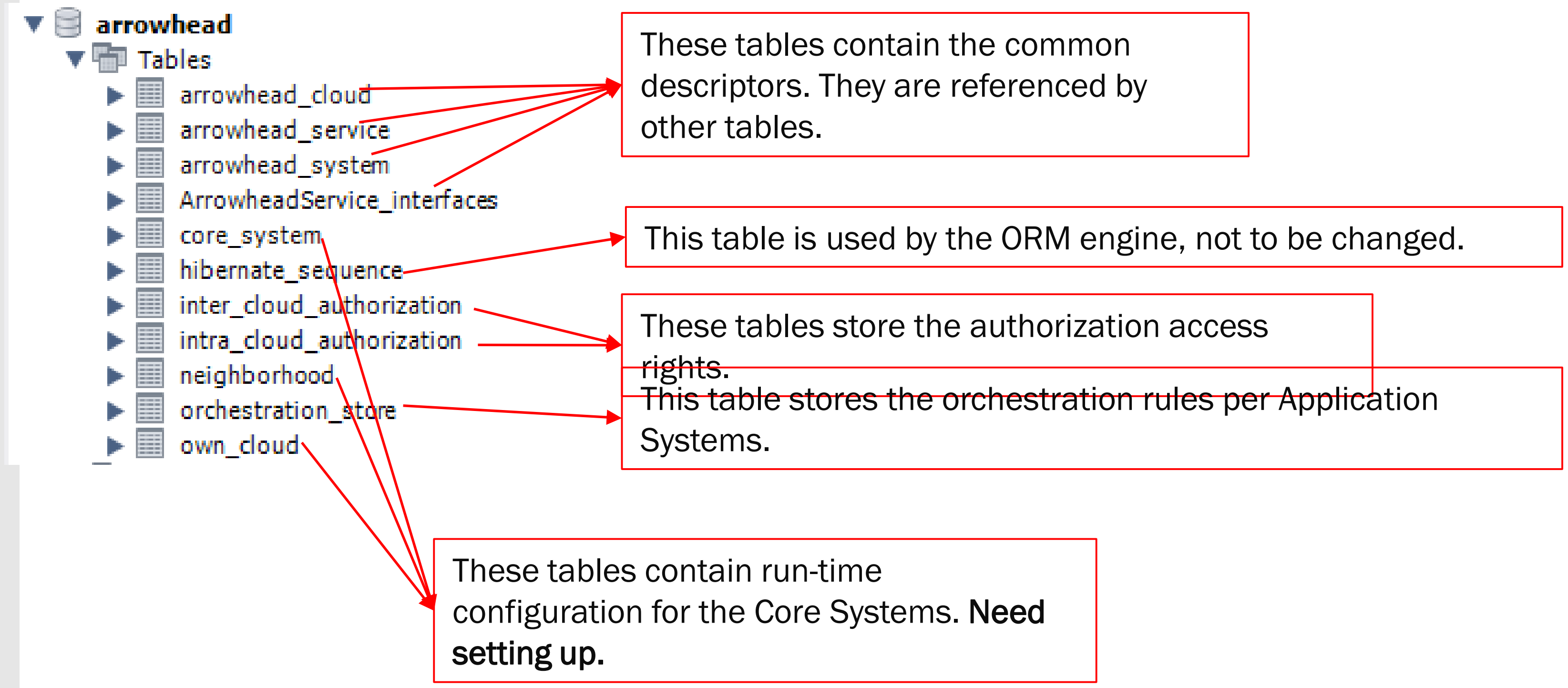
The log4j.properties:



```
# Define the root logger with appender file
log4j.rootLogger = INFO, DB

# Define the DB appender
log4j.appender.DB=org.apache.log4j.jdbc.JDBCAppender

# Set Database Driver
log4j.appender.DB.driver=com.mysql.jdbc.Driver

# Set Database URL
log4j.appender.DB.URL=jdbc:mysql://localhost:3306/arrowhead

# Set database user name and password
log4j.appender.DB.user=root
log4j.appender.DB.password=root

# Set the SQL statement to be executed.
log4j.appender.DB.sql=INSERT INTO LOGS VALUES(DEFAULT,'%m','%C','%d{yyyy-MM-dd HH:mm:ss}','%p')

# Define the layout for file appender
log4j.appender.DB.layout=org.apache.log4j.PatternLayout

# Disable Hibernate verbose logging
log4j.logger.org.hibernate=fatal
```

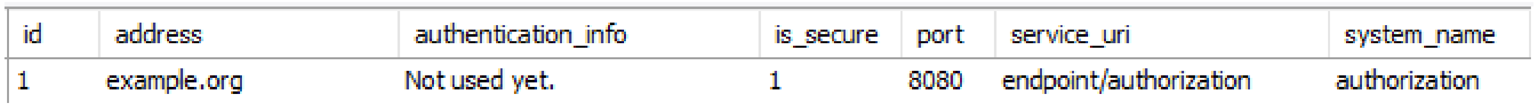
The Arrowhead core database structure

- The joint core database contains the following tables:


arrowhead
Tables
arrowhead_cloud
arrowhead_service
arrowhead_system
ArrowheadService_interfaces
core_system
hibernate_sequence
inter_cloud_authorization
intra_cloud_authorization
neighborhood
orchestration_store
own_cloud
These tables contain the common descriptors. They are referenced by other tables.
This table is used by the ORM engine, not to be changed.
These tables store the authorization access rights.
This table stores the orchestration rules per Application Systems.
These tables contain run-time configuration for the Core Systems. Need setting up.

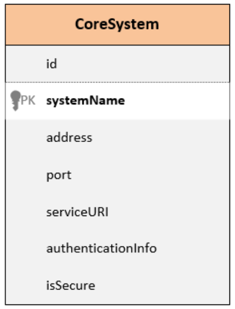
Configuration tables

- These tables can be modified by using the **API module.**, or directly, for example via phpmyadmin.
- The “CoreSystem” table stores where each core system can be reached. This is used by Core Systems for locating other Core Systems.
- For example, let us consider a configuration with the Authorization system started as https://example.org:8080/authorization/ :
  - The system_name field is fixed, and should not to be changed.
  - The ***service_uri*** contains the full URL path (after the port number)
  - The ***is_secure*** boolean is indicating that the Authorization System is started in secure mode, with SSL communication
  - There are hardcoded URL subpaths in the java modules, e.g. „/authorization”, in accordance with the settings in the app.properties file:
    - *base_uri_secured=https://0.0.0.0:8080/endpoint*

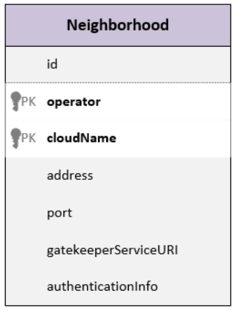


id	address	authentication_info	is_secure	port	service_uri	system_name
1	example.org	Not used yet.	1	8080	endpoint/authorization	authorization

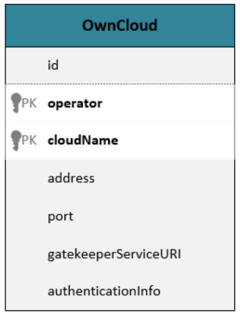
- The ”neighborhood” table references trusted Arrowhead Local Clouds, and the “OwnCloud” table holds information about the Local Cloud itself. These data are used by the Gatekeeper in the inter-Cloud orchestration process.



	CoreSystem
	id
PK	systemName
	address
	port
	serviceURI
	authenticationInfo
	isSecure



	Neighborhood
	id
PK	operator
PK	cloudName
	address
	port
	gatekeeperServiceURI
	authenticationInfo



	OwnCloud
	id
PK	operator
PK	cloudName
	address
	port
	gatekeeperServiceURI
	authenticationInfo

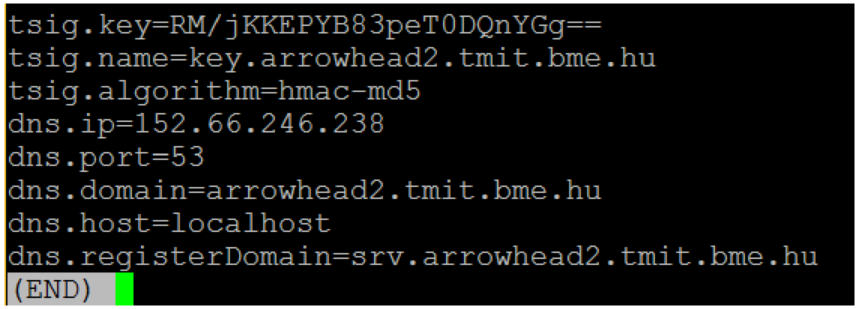
Deploying modules

- Each module can be deployed by running the appropriate JAR file.
- Every module has its own „config" and „lib" folders.
- Insecure (plain HTTP) deployment on the console:
  - *java –jar modulename.jar*
- Secure (SSL) deployment:
  - *java –jar modulename.jar secure*
- Running both (insec and sec) version of the module at the same time:
  - *java –jar modulename.jar both*
- This is valid for all modules.
  - However, the Service Registry requires further configuration.

Deploying the Service Registry REST bridge 1/2

- The Service Registry’s functionalities are based on the access to a properly configured DNS-SD BIND server.
- Configuration of a DNS server is not in scope of this guide.
- If deploying a local DNS server is not possible (e.g.: too time consuming), then the publicly available BME testclouds’ Service Registries can be used temporarily.
  - The „core_system” table has to be set accordingly:
    - arrowhead.tmit.bme.hu:8444/serviceregistry
      (unsecure, server TSIG: „RM/jKKEPYB83peT0DQnYGg==”)
    - arrowhead2.tmit.bme.hu:8444/serviceregistry
      (unsecure, server TSIG: „RM/jKKEPYB83peT0DQnYGg==”)

- The communications with the DNS server is configurable in the „dns.properties” within the SR module, for example:

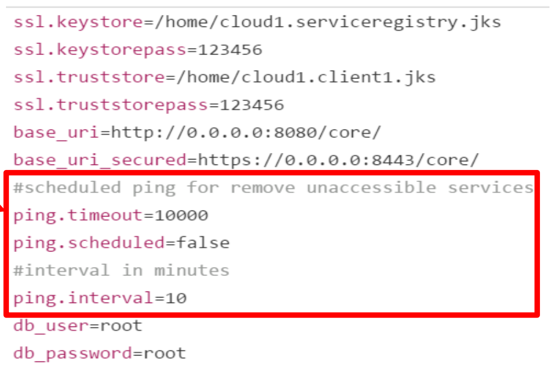


```
tsig.key=RM/jKKEPYB83peT0DQnYGg==
tsig.name=key.arrowhead2.tmit.bme.hu
tsig.algorithm=hmac-md5
dns.ip=152.66.246.238
dns.port=53
dns.domain=arrowhead2.tmit.bme.hu
dns.host=localhost
dns.registerDomain=srv.arrowhead2.tmit.bme.hu
(END)
```

Deploying the Service Registry REST bridge 2/2

- The Service Registry bridge is also capable of cleaning up the DNS server periodically, if enabled in the „app.properties” file. For now, please disable it.

- To use the examples provided in this guide, dummy Service Providers have to be registered in the SR as well.
  - For example, by direct REST interaction using Postman:



```
ssl.keystore=/home/cloud1.serviceregistry.jks
ssl.keystorepass=123456
ssl.truststore=/home/cloud1.client1.jks
ssl.truststorepass=123456
base_uri=http://0.0.0.0:8080/core/
base_uri_secured=https://0.0.0.0:8443/core/
#scheduled ping for remove unaccessible services
ping.timeout=10000
ping.scheduled=false
#interval in minutes
ping.interval=10
db_user=root
db_password=root
```

HTTP/POST:
.../serviceregistry/energy/ChargingReservations/JSON

```
{
 "provider":
 {
  "systemGroup":"evopro_systems",
  "systemName":"server1",
  "address":"address1",
  "port":"1",
  "authenticationInfo":"info1"
 },
 "serviceURI":"/charging_reserv",
 "serviceMetadata":
 [
  {"key":"color","value":"black"}
 ],
 "tSIG_key":"RM/jKKEPYB83peT0DQnYGg==",
  "version":"1.0"
}
```

HTTP/POST:
.../serviceregistry/energy/ChargingReservations/JSON

```
{
 "provider":
 {
  "systemGroup":"evopro_systems",
  "systemName":"server2",
  "address":"address2",
  "port":"1",
  "authenticationInfo":"info2"
 },
 "serviceURI":"/charging_reserv",
 "serviceMetadata":
 [
  {"key":"color","value":"white"}
 ],
 "tSIG_key":"RM/jKKEPYB83peT0DQnYGg==",
  "version":"1.0"
}
```

HTTP/POST:
.../serviceregistry/energy/ChargingReservations/JSON

```
{
 "provider":
 {
  "systemGroup":"evopro_systems",
  "systemName":"server4",
  "address":"address4",
  "port":"1",
  "authenticationInfo":"info4"
 },
 "serviceURI":"/charging_reserv",
 "serviceMetadata":
 [
  {"key":"color","value":"green"}
 ],
 "tSIG_key":"RM/jKKEPYB83peT0DQnYGg==",
  "version":"1.0"
}
```

HTTP/POST:
.../serviceregistry/energy/billing/JSON

```
{
 "provider":
 {
  "systemGroup":"evopro_systems",
  "systemName":"server1",
  "address":"address1",
  "port":"1",
  "authenticationInfo":"info1"
 },
 "serviceURI":"/billing",
 "serviceMetadata":
 [
  {"key":"color","value":"black"}
 ],

"tSIG_key":"RM/jKKEPYB83peT0DQnYGg==",
  "version":"1.0"
}
```

Setting up a secondary Local Cloud for testing inter-Cloud orchestration

- To test out inter-Cloud orchestration, the secondary Cloud can be set up similarly.
  - Please import the „create_arrowhead_db_2.sql" script
  - Modify the „address" field in the „own_cloud" table for both databases.
  - Also modify the „address" field in the „neighborhood" table in Cloud 1 so the Gatekeeper can contact Cloud 2
  - Post the following entries in Cloud 2's Service Registry (The BME testclouds have these.)

HTTP/POST:	HTTP/POST:
.../serviceregistry/energy/ChargingReservations/JSON	.../serviceregistry/energy/chargingType/JSON

```
{
 "provider":
 {
  "systemGroup":"AUDI_systems",
  "systemName":"server1",
  "address":"address1",
  "port":"1",
  "authenticationInfo":"info1"
 },
 "serviceURI":"/charging_reserve",
 "serviceMetadata":
 [
  {"key":"color","value":"green"}
 ],
 "tSIG_key":"RM/jKKEPYB83peT0DQnYGg==",
 "version":"1.0"
}
```

```
{
 "provider":
 {
  "systemGroup":"AUDI_systems",
  "systemName":"server1",
  "address":"address1",
  "port":"1",
  "authenticationInfo":"info1"
 },
 "serviceURI":"/charge_type",
 "serviceMetadata":
 [
  {"key":"color","value":"green"}
 ],
 "tSIG_key":"RM/jKKEPYB83peT0DQnYGg==",
  "version":"1.0"
}
```

Using the examples

- We provided one scenario to experiment with the Advanced Orchestration Service of Arrowhead.
- These are consistent with the dummy data imported in the databases by executing the ***create_arrowhead_database_1 & 2*** scripts .
- These scenarios include sending Service Requests to the Orchestrator and receiving Orchestration Response based on the dummy data.

- Recommendation: test these configuration manually, interacting through plain HTTP.
  – Example: using the **Postman** development test tool (Google Chrome extension)
  – https://www.getpostman.com/

Use case scenario for manual testing

- The test scenario regards an automotive case. There are two Local Clouds defined:
  - Cloud 1 belongs to a charging infrastructure owner with charging stations and their management systems (servers). Here, evopro Inc. owns this deployment.
  - Cloud 2 belongs to a car manufacturer and it includes electric cars that can look for charging services. Here, AUDI is brought as an example. This Cloud also has a charging reservation server operating.

- There are multiple Application Services defined for testing out orchestration. Some Service Providers are missing qualifications (e.g. missing authorization or simply are offline).
  - ChargingReservation can be requested by „charging_stations" and server[s] are capable of serving it (either in Cloud 1 or 2).
  - The ChargingType service can requested by cars (from AUDI's cloud) and charging_stations (from evopro's) are capable of serving it. It would define what type of charging the given station is capable of.
  - The „billing" service can be requested by charging_stations and evopro's Management Servers are capable of serving it.
  - Systems in AUDI's cloud can request DC_charging service and evopro's charging_stations are capable of providing it.

- These are shown on the next slide.

Service 1

ServiceGroup: energy
ServiceDefinition: billing
Interfaces: JSON
Service Metadata: color:black

Service 2

ServiceGroup: energy
ServiceDefinition: chargingType
Interfaces: JSON
Service Metadata: color:green

Evopro Cloud

Operator: evopro
CloudName: evopro_cloud
Gatekeeper IP, Port, URI:
... :8446/gatekeeper
AuthenticationInfo: NotUsed

Service 3

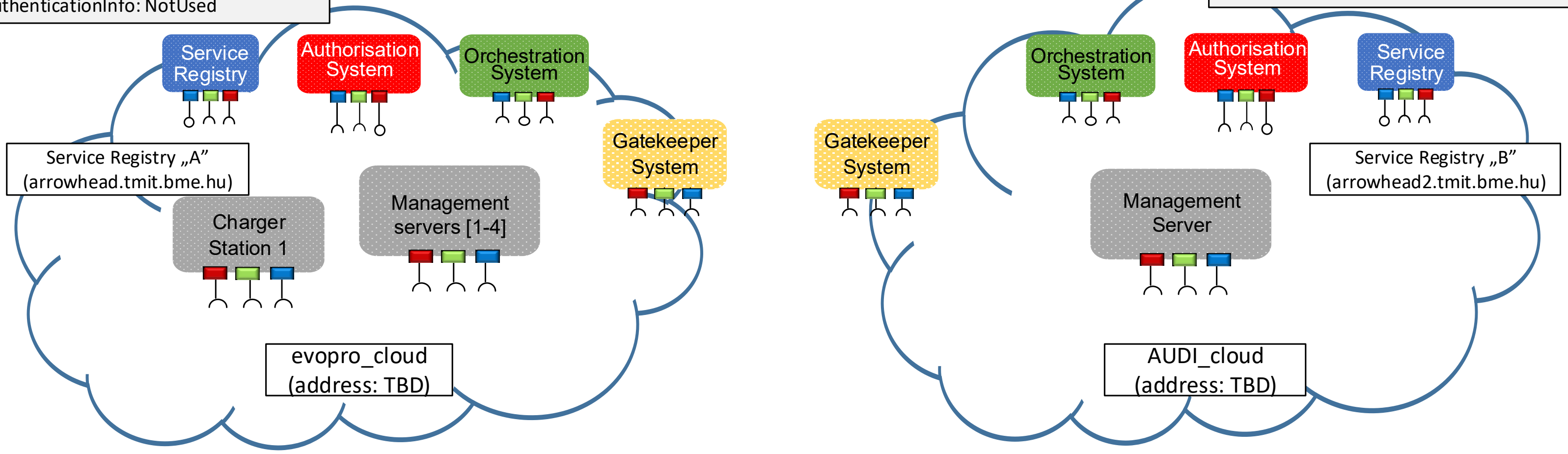
ServiceGroup: charging
ServiceDefinition: DCcharging
Interfaces: JSON
Service Metadata: voltage:20kV

Service 4

ServiceGroup: energy
ServiceDefinition: ChargingReservations
Interfaces: JSON
Service Metadata: color:green/black/white

AUDI Cloud

Operator: AUDI
CloudName: AUDI_cloud
Gatekeeper IP, Port, URI:
... :8446/gatekeeper
AuthenticationInfo: NotUsed



ArrowheadSystem 1 (Cloud Evopro)

SystemGroup: evopro_car_chargers
SystemName: charging_station1
Address: 0.0.0.0
Port: 8080
AuthenticationInfo: evocert1

ArrowheadSystem 2-5 (Cloud Evopro)

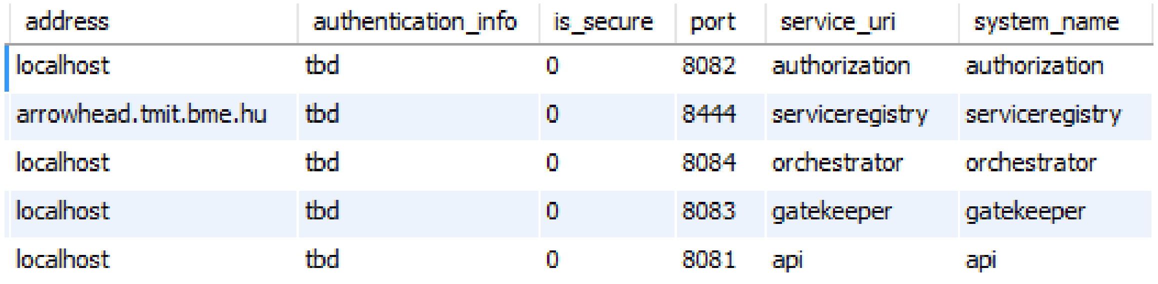
SystemGroup: evopro_systems
SystemName: server1-4
Address: address1-4
Port: 1
AuthenticationInfo: evocert2-5

ArrowheadSystem 6 (Cloud AUDI)

SystemGroup: AUDI_systems
SystemName: server1
Address: address1
Port: 1
AuthenticationInfo: audicert1

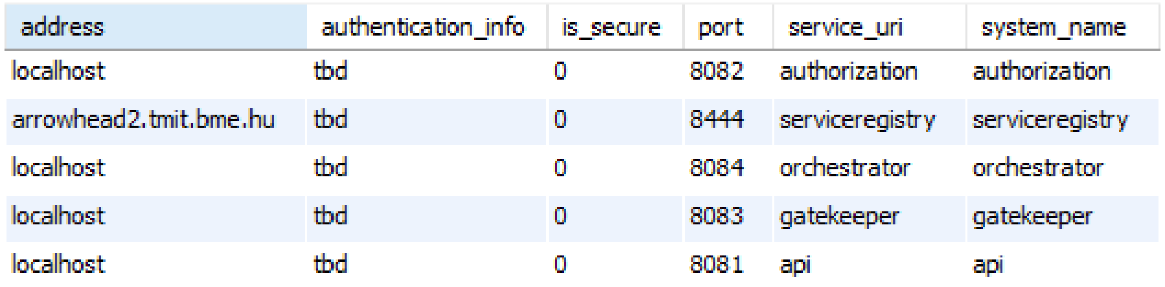
Testbed servers deployment information

- Cloud1: arrowhead.tmit.bme.hu



address	authentication_info	is_secure	port	service_uri	system_name
localhost	tbd	0	8082	authorization	authorization
arrowhead.tmit.bme.hu	tbd	0	8444	serviceregistry	serviceregistry
localhost	tbd	0	8084	orchestrator	orchestrator
localhost	tbd	0	8083	gatekeeper	gatekeeper
localhost	tbd	0	8081	api	api

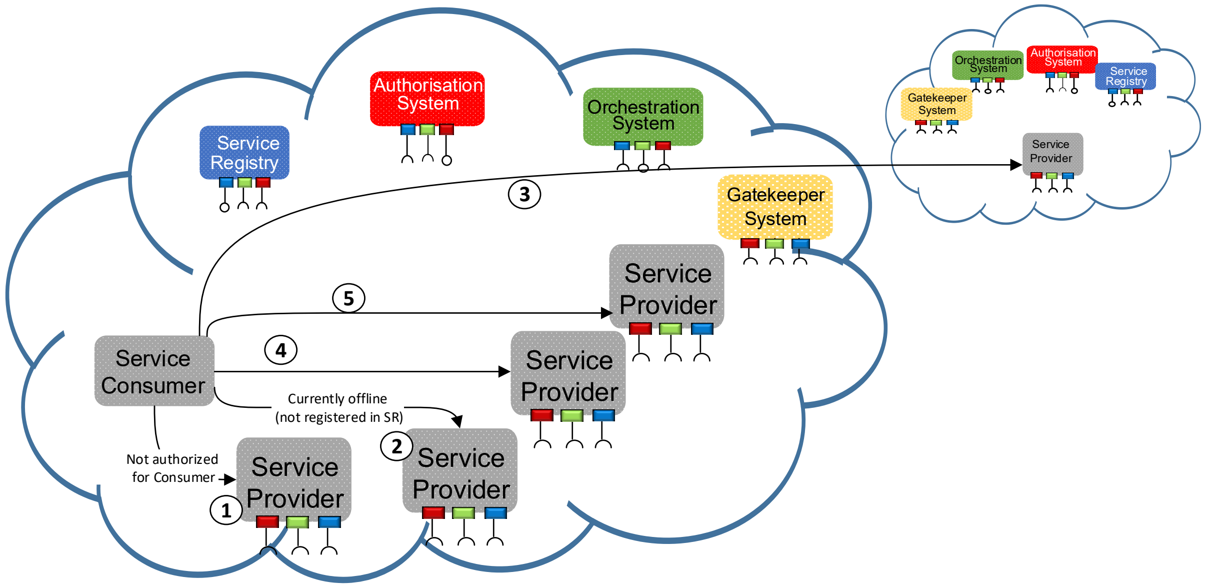
- Cloud2: arrowhead2.tmit.bme.hu



address	authentication_info	is_secure	port	service_uri	system_name
localhost	tbd	0	8082	authorization	authorization
arrowhead2.tmit.bme.hu	tbd	0	8444	serviceregistry	serviceregistry
localhost	tbd	0	8084	orchestrator	orchestrator
localhost	tbd	0	8083	gatekeeper	gatekeeper
localhost	tbd	0	8081	api	api

Store-based orchestration test

- Charging_stations from evopro's Cloud (#1) are hardwired to use a certain list of Management Servers.
- However, which station can access which server is dynamic (e.g. changes with time of day)
  - Some servers go offline or night-time management of stations belongs to an external party
- These orchestration rules are stored and iterated through based on their priority. Check Advanced Orchestration Service SD



Store-based orchestration

Service Request Form

```
{
 "requesterSystem":
 {
    "systemGroup": "evopro_car_chargers",
    "systemName": "charging_station1",
    "address": "0.0.0.0",
    "port": "8080",
    "authenticationInfo": "info1"
 },
 "requestedService":
 {
    "serviceGroup": "energy",
    "serviceDefinition": "ChargingReservations",
    "interfaces": ["JSON"]
 }
}
```

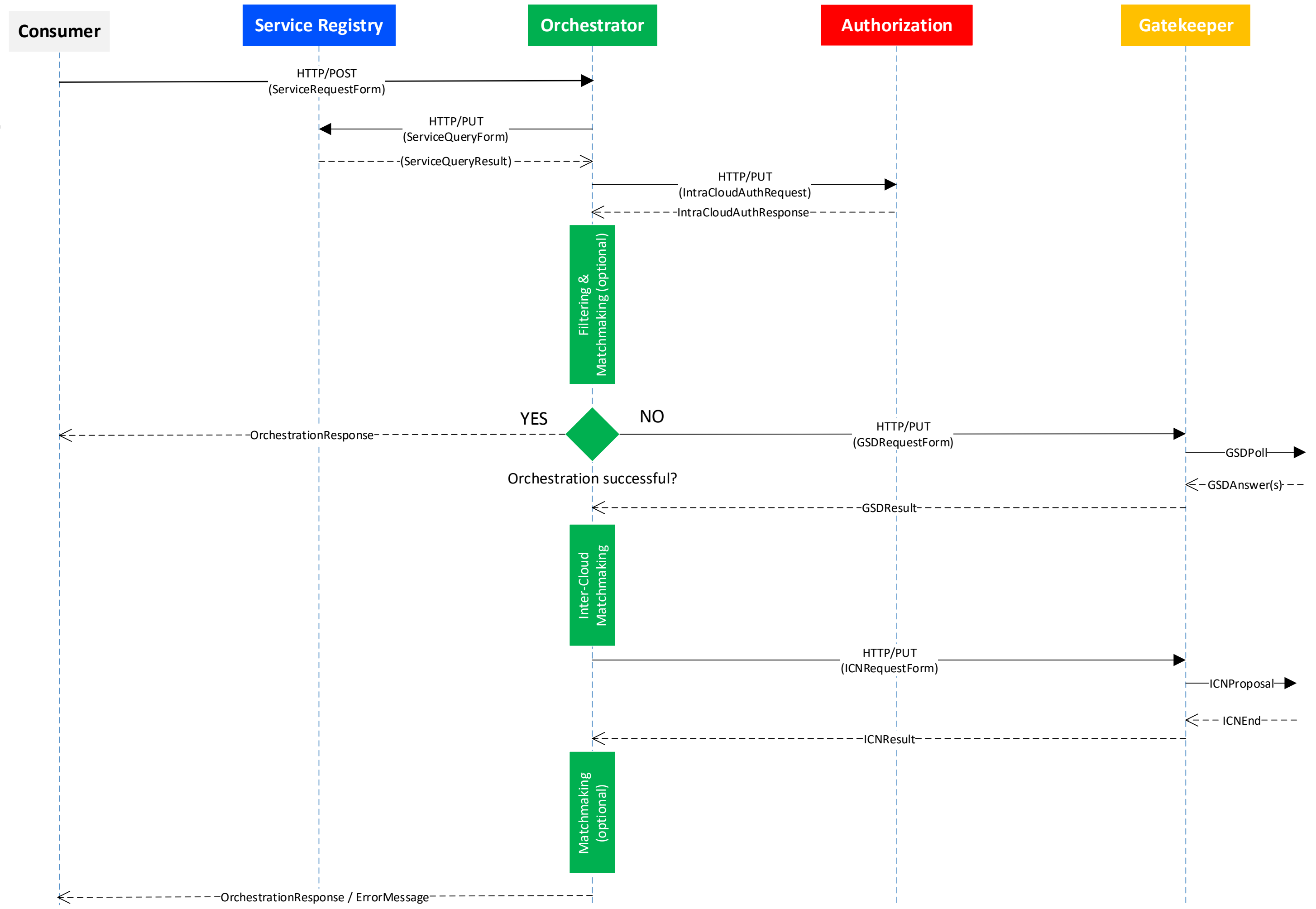
Orchestration Response (expected)

```
{
 "response": [
   {
    "instruction": "/charging",
    "provider": {
      "address": "address2",
      "authenticationInfo": "info2",
      "port": "1",
      "systemGroup": "evopro_systems",
      "systemName": "server2"
    },
    "service": {
      "interfaces": [
        "JSON"
      ],
      "serviceDefinition": "ChargingReservations",
      "serviceGroup": "energy"
    }
   }
 ]
}
```

In this example the Orchestrator will iterate through the 5 store entries in priority order. (All store entries are for this consumer/service pair.) The 1st entry is not registered in the Authorization, while the 2nd entry is not registered in the Service Regsitry. The 3rd entry is the inter-cloud entry which should be the response if the 2nd cloud is properly configured. Otherwise the response will contain the 4th (local) provider.

Checking the dynamical orchestration process

The flowchart describes what happens / what you must do in the example on dynamical orchestration capabilities.



Consumer
Service Registry
Orchestrator
Authorization
Gatekeeper
HTTP/POST (ServiceRequestForm)
HTTP/PUT (ServiceQueryForm)
(ServiceQueryResult)
HTTP/PUT (IntraCloudAuthRequest)
IntraCloudAuthResponse
Filtering & Matchmaking (optional)
YES
NO
OrchestrationResponse
HTTP/PUT (GSDRequestForm)
Orchestration successful?
GSDPoll
GSDAnswer(s)
GSDResult
Inter-Cloud Matchmaking
HTTP/PUT (ICNRequestForm)
ICNProposal
ICNEnd
ICNResult
Matchmaking (optional)
OrchestrationResponse / ErrorMessage

Local dynamical orchestration (no preferences or matchmaking)

Service Request Form

```
{
 "requesterSystem":
 {
    "systemGroup": "evopro_car_chargers",
    "systemName": "charging_station1",
    "address": "0.0.0.0",
    "port": "8080",
    "authenticationInfo": "info1"
 },
 "requestedService":
 {
    "serviceGroup": "energy",
    "serviceDefinition": "ChargingReservations",
    "interfaces": ["JSON"]
 },
 "orchestrationFlags": {
   "entry": [
     {
       "key": "overrideStore",
       "value": true
     }
   ]
 }
}
```

Orchestration Response (expected)

```
{
 "response": [
  {
   "provider": {
    "address": "address2",
    "authenticationInfo": "info2",
    "port": "1",
    "systemGroup": "evopro_systems",
    "systemName": "server2"
   },
   "service": {
    "interfaces": [
     "JSON"
    ],
    "serviceDefinition": "ChargingReservations",
    "serviceGroup": "energy",
    "serviceMetadata": [
     {
      "key": "color",
      "value": "white"
     }
    ]
   },
   "serviceURI": "/charging_reserv"
  },
  {
   "provider": {
    "address": "address1",
    "authenticationInfo": "info1",
    "port": "1",
    "systemGroup": "evopro_systems",
    "systemName": "server1"
   },
   "service": {
    "interfaces": [
     "JSON"
    ],
    "serviceDefinition": "ChargingReservations",
    "serviceGroup": "energy",
    "serviceMetadata": [
     {
      "key": "color",
      "value": "black"
     }
    ]
   },
   "serviceURI": "/charging_reserv"
  }
 ]
}
```

Here we ignore the Orchestration Store, but looking for the same service in the local cloud. Since we did not give any preferences or asked for matchmaking, we get back all the providers that are capable to satisfy this service request. These providers are registered in the Service Registry and also authorized.
Here, charging stations are capable of checking their reservations in all currently available servers.

Inter-Cloud dynamical orchestration (with triggerInterCloud flag)

Service Request Form

```
{
 "requesterSystem":
 {
    "systemGroup": "evopro_car_chargers",
    "systemName": "charging_station1",
    "address": "0.0.0.0",
    "port": "8080",
    "authenticationInfo": "info1"
 },
 "requestedService":
 {
    "serviceGroup": "energy",
    "serviceDefinition": "ChargingReservations",
    "interfaces": ["JSON"]
 },
 "orchestrationFlags": {
   "entry": [
     {
       "key": "triggerInterCloud",
       "value": true
     }
   ]
 }
}
```

Orchestration Response (expected)

```
{
 "response": [
  {
   "provider": {
    "address": "address1",
    "authenticationInfo": "info1",
    "port": "1",
    "systemGroup": "AUDI_systems",
    "systemName": "server1"
   },
   "service": {
    "interfaces": [
     "JSON"
    ],
    "serviceDefinition": "ChargingReservations",
    "serviceGroup": "energy",
    "serviceMetadata": [
     {
      "key": "color",
      "value": "green"
     }
    ]
   },
   "serviceURI": "/charging_reserv"
  }
 ]
}
```

Looking for the same service again, but this time we only ask for non-local provider systems. If the 2nd cloud is configured, we get back the only registered provider for this service. This answer could also have more providers, since we did not ask for matchmaking.

Dynamical orchestration with Inter-Cloud enabled (requires secondary Local Cloud configured)

Service Request Form

```
{
 "requesterSystem":
 {
    "systemGroup": "evopro_car_chargers",
    "systemName": "charging_station1",
    "address": "0.0.0.0",
    "port": "8080",
    "authenticationInfo": "info1"
 },
 "requestedService":
 {
    "serviceGroup": "energy",
    "serviceDefinition": "chargingType",
    "interfaces": ["JSON"]
 },
 "orchestrationFlags": {
   "entry": [
     {
      "key": "overrideStore",
      "value": true
     },
     {
      "key": "enableInterCloud",
      "value": true
     }
   ]
 }
}
```

Orchestration Response (expected)

```
{
 "response": [
  {
   "provider": {
    "address": "address1",
    "authenticationInfo": "info1",
    "port": "1",
    "systemGroup": "AUDI_systems",
    "systemName": "server1"
   },
   "service": {
    "interfaces": [
     "JSON"
    ],
    "serviceDefinition": "chargingType",
    "serviceGroup": "energy",
    "serviceMetadata": [
     {
      "key": "color",
      "value": "green"
     }
    ]
   },
   "serviceURI": "/charge_type"
  }
 ]
}
```

In this final example we ask for a different service, which can only be found in the 2nd cloud. First the Orchestrator tries to find a local provider system and when that fails, it asks the neighborhood clouds too.
Here, charging stations are announcing their charging capabilities within all mgmt. servers where they can.

Outline

- Setup of Raspberry PI
- Arrowhead installing
- Arrowhead programming

Getting started with the App. System skeleton

- There are three client skeletons available in the distribution:
  - Service Provider module (without SSL support): registers in SR, offers REST resource, unregisters from SR upon exit
  - Service Consumer (without Java Jersey library or SSL support): capable of requesting orchestration and based on that connecting to a running Service Provider skeleton to retrieve dummy temperature service information.
  - Service Consumer Client with SSL support (based on Jersey-client library): same capability, but uses SSL to connect to the Orchestrator
- Also available at: https://github.com/hegeduscs/arrowheadclient

Executing the App. System skeleton

- The three skeletons are maven projects, importable for example in Eclipse
  - The code features hardwired Strings that contain configuration
  - They do not need any further configuration
- The service interactions of the skeleton demos are configured (authorized) in the Core Systems database scripts.
  - Therefore, testable without further configuration.

Next steps

- Configuring the Core Systems
  - For example, by direct interaction with the core database
- Enabling SSL and use of certificates