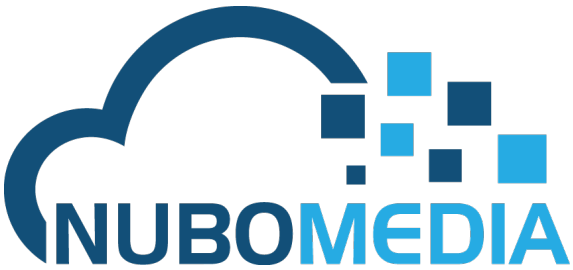
D3.5.1	
Version	1.0
Author	Fraunhofer
Dissemination	PU
Date	27/01/2015
Status	Final



D3.5.1: NUBOMEDIA Signaling Plane

Project acronym:	NUBOMEDIA
Project title:	NUBOMEDIA: an elastic Platform as a Service (PaaS) cloud for interactive social multimedia
Project duration:	2014-02-01 to 2016-09-30
Project type:	STREP
Project reference:	610576
Project web page:	http://www.nubomedia.eu
Work package	WP3: NUBOMEDIA Cloud Platform
WP leader	TUB
Deliverable nature:	Other
Lead editor:	Fraunhofer
Planned delivery date	01/2015
Actual delivery date	27/01/2015
Keywords	Signaling, Cloud Middleware, Application, Clustering

The research leading to these results has been funded by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement nº 610576

FP7 ICT-2013.1.6. Connected and Social Media

Contributors:

Niklas Blum (Fraunhofer FOKUS)
Frank Schulze (Fraunhofer FOKUS)

Internal Reviewer(s):

Luis López (URJC)

Version History

Version	Date	Authors	Comments
0.1	26.06.2014	Niklas Blum (Fraunhofer) Frank Schulze (Fraunhofer)	
0.2	23.12.2014	Frank Schulze (Fraunhofer)	
1.0	16.01.2015	Fraunhofer	Added chapters about -State of the art, -Setup&Configuration -Module development

Table of contents

List of Figures:

Acronyms and abbreviations:

IMS	IP Multimedia Subsystem
RTC	Real Time Communication
M2M	Machine-To-Machine
OpenXSP	Open Extensible Service Platform
JSON	JavaScript Object Notation
RPC	Remote Procedure Call
XMPP	Extensible Messaging and Presence Protocol
PAPI	NUBOMEDIA Platform API
DAPI	NUBOMEDIA Development API
UA	User Agent
JDK	Java Development Kit
SIP	Session Initiation Protocol
P-CSCF	Proxy Call Session Control Function
S-CSCF	Service Call Session Control Function
I-CSCF	Interrogating Call Session Control Function
CLI	Command Line Interface
IDE	Integrated Development Environment
WS	WebSocket

1 Executive Summary

This document describes how the NUBOMEDIA Signaling Plane will be designed. The Signaling Plane will be based on OpenXSP, a middleware which provides a framework for developing loosely coupled and cloud enabled network service applications. OpenXSP is developed at Fraunhofer FOKUS based on Vert.x IO[1].
Vertx.io is a polyglot framework, meaning that applications can be written in a number of languages such as Java, JavaScript, CoffeeScript, Ruby, Python or Groovy.

The NUBOMEDIA Signaling Plane will consist of the following components:

- An **Event Bus** as message distribution layer to assure that elements can communicate in a distributed environment (without "knowing" that they are distributed)
- **NUBOMEDIA Modules** that provide the NUBOMEDIA functional logic (e.g. Application Function, Connectors and other components that communicate with them)
- **Event Bus protocol** which is used by modules to interact with each other in order to create sessions

2 State Of The Art

The purpose of this section is to prepare the development of a service platform upon which the NUBOMEDIA Signaling Plane will be built on. In order not to build up everything from the ground, the platform should be based on some existing framework. The main scope of this section is to analyze and choose such a platform. Consequently, this section is divided into three parts:

1. Gather requirements
2. Gather candidates for frameworks
3. Map requirements to frameworks

2.1 Requirements

The following features are considered a must-have.

- **Cloud deployable**
  It must be possible to deploy applications written for the platform in cloud computing environments. That especially means that it must be possible to create clusters of application instances.
  Rationale: Cloud computing deployments are mandatory for cloud platforms like NUBOMEDIA

- **Java support**
  It must be possible to write applications (not the framework itself) in Java.
  Rationale: Java is one of the most used languages across the board. Developers know it, operators love it. It boasts a rich eco-system and broad platform support. Supporting Java will make it easy for most new developers to get on board and start developing.

- **Python support**
  It must be possible to write applications (not the framework itself) in Python.
  Rationale: Python is the most popular dynamically typed language. A very stringent object system and a lot of syntactic sugar make it very easy to rapidly develop applications in Python. Like Java, it boasts a rich eco-system and broad platform support.

- **Asynchronous I/O**
  The platform must support asynchronous (non-blocking I/O). That means it must be possible to service a number of client requests concurrently without threading/forking.
  Rationale: The majority of applications to run on the envisioned platform are foreseen to be I/O intensive - if not I/O bound. For these usage scenarios, threads and processes as the main method for concurrency are a waste of resources that inhibits scalability

- **Blocking semantics**
  Asynchronous I/O platforms tend to impose a callback driven style upon developers that can be tough to wrap your head around (e.g. like in NodeJS). Apart from code that is closely tied to the network, it must be possible to use traditional programming styles.
  Rationale: A code full of callbacks is more difficult to maintain [4]

- **Background tasks**
  Running computational intensive tasks inside an asynchronous event loop is unfeasible. It must be possible to defer those tasks to background workers (threads, processes, co-routines etc.)
  Rationale: CPU intensive tasks are not excluded an would block the event bus[5]

- **Event Bus**
  The platform must support a distributed event bus that spans multiple instances of the platform.
  Rationale: A distributed event bus replaces the tight coupling of components with a flexible loosely coupled design.

- **Configuration Framework**
  The platform must support some form of a configuration framework
  Rationale: Every software needs to be configured. It's time to consuming to redo the work for this over and over again. Furthermore, a common configuration framework means that all applications will be configured in a similar manner, providing for easier maintenance.

- **Logging**
  The platform must support some form of a logging framework.
  Rationale: See Configuration

- **Raw Sockets**
  To be able to read from non-IP sockets for streams of data

- **Debugging**
  Easy to debug very useful because testing and debugging is 60% of work

- **Memory management**
  To have a garbage collector
  Rationale: MM is hard to implement, but important to avoid things such as memory leaks. Out of the box GC will make life easier.

- **Scalability**
  Ability to use more than one CPU when having more than one thread/process.
  Ability to service multiple requests without spawning threads/processes.

The following features are considered a **nice-to-have:**

- **JavaScript support**
  It would be nice to be able to write applications (not the framework itself) in JavaScript.
  Rationale: Some people actually like JS.

- **C/C++ support**
  It would be nice to be able to write applications (not the framework itself) in C/C++.
  Rationale: For high-performance, C is sometimes the way to go

2.2 Existing Frameworks

The following frameworks have been evaluated as potential candidates:

- **Node.js**
  Node.js[2] is a software platform that is used to build scalable network (especially server-side) applications. Node.js utilizes JavaScript as its scripting language, and achieves high throughput via non-blocking I/O and a single-threaded event loop.

- **Gevent**
  gevent[3] is a coroutine-based Python networking library that uses greenlet to provide a high-level synchronous API on top of the libev event loop.

- **OpenXSP/Vert.x**
  OpenXSP is a PaaS middleware developed at Fraunhofer FOKUS. It has already been as basis for Real Time Communication (RTC) and M2M service platforms. OpenXSP is in turn based on Vert.x IO and inherits a lot of its characteristics. Vert.x[1] is a lightweight, high performance, polyglot and event-driven application framework that runs on the Java Virtual Machine.

- **Wharf**
  A flexible and powerful prototyping platform for NGN components developed at Fraunhofer FOKUS. It is written in C and C++.

2.3 Feature Comparison

Color Key:

- Supported out-of-the-box
- Unsupported but doable without too much effort
- Unsupported and hard to do or lot of work
- Impossible / Unfeasible

Must haves:

	Node.js	gevent	OpenXSP	Wharf
Cloud deployable				
Java support				
Python support				
Aynchronous I/O				
Blocking semantics				
Background tasks				
Event Bus				
Configuration framework				
Logging				
Raw Sockets				

Debugging				
Memory Management (Garbage Collector GC)				
Scalability				

Nice-to-haves:

	Node.js	**gevent**	**OpenXSP**	**Wharf**
C/C++ support				
Javascript support				

2.4 Conclusion

As can be seen, none of the frameworks fulfill all requirements. OpenXSP though is corresponding to a lot of the requirements. That's why it will be used as the basic framework for the development of the NUBOMEDIA Signaling Plane.

3 The Event Bus

3.1.1 Overview

The Event Bus is the nervous system of the Signaling Plane. It allows modules of the Signaling Plane to communicate with each other irrespective of what language they are written in, and whether they are in the same OpenXSP instance, or in a different OpenXSP instance.
It even allows client side JavaScript running in a browser to communicate on the same Event Bus.

The Event Bus forms a distributed peer-to-peer messaging system spanning multiple server nodes and multiple browsers. It can be extended dynamically through several virtual and physical machines.

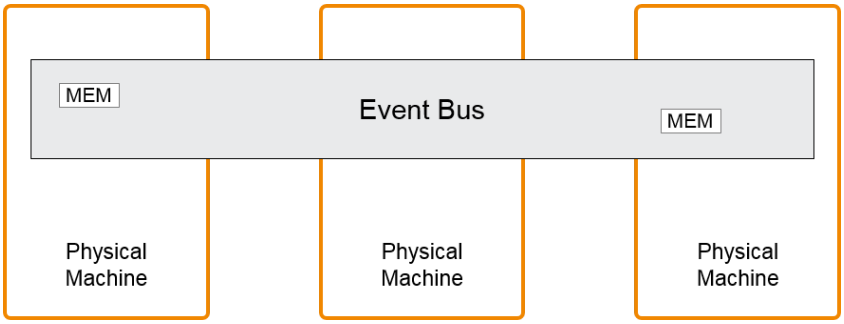


Figure 1. Distributed Event Bus

3.1.2 Addressing

Messages are sent on the event bus to an address. OpenXSP and thus the Signaling Plane is not bound to any addressing schemes. An address is simply a string, any string is valid. Some examples of valid addresses are `europe.news.feed1`, `acme.games.pacman`, `sausages`, and `X`.

As a convention the names of the packages that implement certain functionalities should also be represented on the event bus and should be combined with a meaningful event/operation name, e.g. `org.acme.MyPackage.MyClass.doSomething`.

3.1.3 Handlers

A handler is an entity that receives messages from the bus. You register a handler at an address. Many different handlers from the same or different modules can be registered at the same address. A single handler can be registered at many different addresses at the same time.

3.1.4 Messaging Schemes

The Event Bus supports the following modes of operation:

- Publish / subscribe messaging
  Publishing means delivering the message to all handlers that are registered at that address. This is the familiar publish/subscribe messaging pattern.

- Point to point and Request-Response messaging
  Messages are routed to just one of the handlers registered at an address. They can optionally be replied to.
- Remote Procedure Call (RPC)
  This mode of operation is implemented by OpenXSP on top of the Request-Response model, basically by enforcing certain conventions on requests and responses.

3.1.5 Types of Messages

Messages that you send on the event bus can be as simple as a string, a number or a boolean. It is also possible to send Vert.x buffers or JSON messages.
It's highly recommended to use JSON messages to communicate between verticles. JSON is easy to create and parse in all the languages that OpenXSP supports.
For RPC messages, JSON is enforced.

4 NUBOMEDIA Modules

This section will describe in more detail how modules on the Signaling Plane communicate with each other using the event based middleware provided by OpenXSP and how these modules are supposed to be designed.
The NUBOMEDIA Modules implement the specific NUBOMEDIA functionalities like Media Elements, Connectors and API Modules.

4.1 Basics

The unit of execution for OpenXSP applications is called a "Verticle".
The unit of packaging for OpenXSP applications is called a "Module". A module can contain any number of (including zero) verticles and can depend on other modules (and their verticles) in turn. Creating a module with no verticles makes sense to provide only library support for other modules. Modules are described by a descriptor file: mod.json. A minimal descriptor looks like this:

```
{
  "owner": "org.acmecorp",
  "name": "myNobomediaAdapterModule",
  "version": "0.1"
}
```

Additionally, three more fields are optionally recognized:

- `worker`
  indicates if this is a worker module. See below under event loop.
- `main`
  Indicates the startup routine for this module.
- `includes`
  Additional module dependencies as a comma-separated string.

The unit of deployment for OpenXSP applications is called an "app". An app includes one or more modules and their configuration and runs them within one OpenXSP instance. Note however, that this is only feasible for simple deployments where it is desirable to run all modules of an application in one instance (and hence in one machine).

4.2 The Event Loop

By default, all verticles run in an asynchronous event loop. When developing a verticle, it is essential not to block the event loop. Blocking here means either doing any kind of blocking I/O or even doing any kind of computational intensive work. Modules that do either of these should indicate that they are so called "worker" modules by setting "worker": true in their mod.json file.
The advantage of an event loop is that it is enormously scalable. Instead of waiting for I/O operations to complete, the executing thread will rather do other stuff (e.g. servicing the next request) in the meantime. This is achieved by using a callback driven style of programming. Imagine the following scenario:

- We want to read some data in an I/O intensive operation (function `readData`)
- We want to do something with that data (function `doSomething`)
- We want to do something completely different (function `doSomethingUnrelated`)

In the traditional blocking world we would do something like the following:

```
def doSomething(data):
    # do something with data
      data = readData()
      doSomething(data)
      doSomethingUnrelated()
```

What happens here is the following: After the data is read, the program waits for the operation (`readData`) to complete (which is consuming the event loop thread lifetime). As soon as readData returns, we have our data and can go on to do something with it (`doSomething(data)`). Finally, when that is done, we can go on and do other stuff (`doSomethingUnrelated`).

In the asynchronous world, we do something like this:

```
def doSomething(data):
    # do something with data
      readData(callback = doSomething)
      doSomethingUnrelated()
```

As can be seen, the result of `readData` is not received in the functions return value. Instead `doSomething` is passed in the handler method as a callback. The framework will take care that this handler is called asynchronously as soon as the data is available.

5 Event Bus Protocol

The Event Bus Protocol will enable NUBOMEDIA modules to create sessions. During the design phase of the protocol, special attention is devoted to a very generic design. This allows the translation not only to SIP, but also to other protocols (e.g. XMPP).
The protocol contains mechanisms for basic session management:

- Create sessions
- Update sessions
- Cancel sessions
- End sessions

5.1.1 Creating sessions

The creation of sessions follows a three-way-handshake pattern. This means the initiator sends a session indication to the session participant. This participant sends an accept (or deny) response to the initiator. The initiator confirms the response.

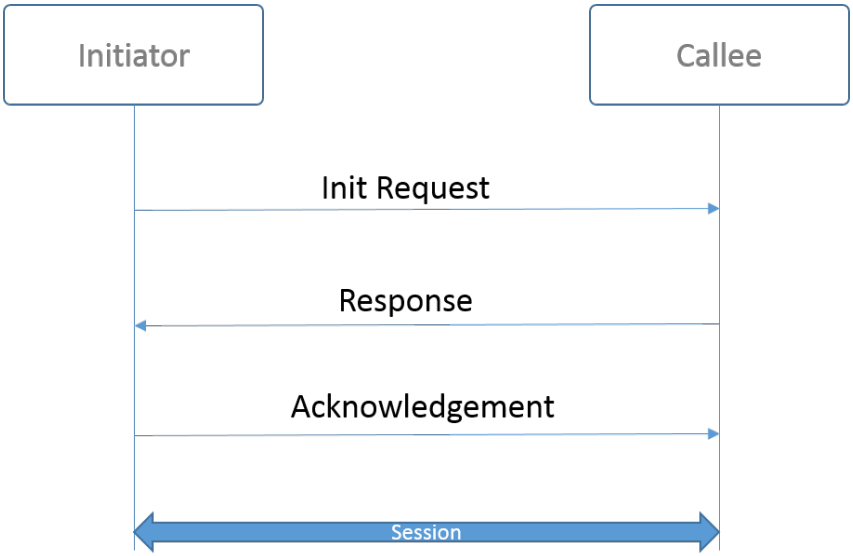


Figure 2 Session Establishment

5.1.2 Requests

The requests are based on JSON strings. The JSON objects contain different fields to indicate the type of request. The most important field is the method-field. This field indicates the type of the message (session initiation request etc.)

Create session request

```
{
        "from":"Alice",
        "to":"Bob",
        "method":"init",
        "session_id":"123456"
        "address":"callback_address_for_responses",
        "content":"Some sdp or other data for session establishment ...",
        "content_type":"application/sdp",
        "header":{
                "User-Agent":"RTC Control",
                "ApplicationSpecificHeader":"header_value"
        }
 }
```

Accept session request

```
{
```

```
        "method":"accept",
        "session_id":"123456",
        "content":"Some sdp ...",
        "content_type":"application\/sdp",
        "address":"callback_address_for_responses",
        "header":
                //some application specific header
        }
}
```

Cancel session initiation request

```
{
        "method":"cancel",
        "session_id":"123456",
        "address":"callback_address_for_responses"
}
```

Confirm session establishment

```
{
        "method":"confirm",
        "session_id":"123456"
        "content":"E.g.       some       sdp       to       update       the       session",
                "content_type":"application\/sdp",
        "address":"callback_address_for_responses"
}
```

Update session request

```
{
        "method":"update",
        "session_id":"123456",
        "content":"Some sdp to update the session",
        "content_type":"application\/sdp",
        "address":"callback_address_for_responses"
}
```

End Session request

```
{
        "method":"end",
        "session_id":"123456",
        "address":"callback_address_for_responses"
}
```

5.1.3 Request Fields

`method` field

This field is mandatory to indicate the type of request. Current supported values are

- init
- accept
- cancel
- confirm
- end
- update

`address` field

This field contains a callback address of the sender in order to allow the recipient of the request to send response requests related to this message. There is no convention of how to generate the value of this header - the sender-application is responsible for this.

`to` field

This field is a human readable value for the identification of the receiver of the message.

`from` field
This field is a human readable value for the identification of the sender of this message.

`session_id` field
This field identifies the session to which the session belongs to so that the receiver can map the request to a session.

`content` field
If this message contains some data, then it should go as value of the content field.

`content_type` field
If the message contains data in the `content` field, then the type of the data needs to be described here

`header` field
This field serves as extension for application specific metadata that might be exchanged additionally between the sender and receiver

6 Functional Modules

6.1 SIP Stack

The SIP stack allows application developers to develop SIP client and server applications. It needs to be integrated as a library module in the mod.json descriptor in order to make use of the SIP stack.

6.2 Application Function

The NUBOMEDIA Application Function provides the server side logic of the NUBOMEDIA Platform API (NUBO-PAPI). It allows clients to initiate application sessions with usage of the NUBOMEDIA media services provided by the NUBOMEDIA media elements.
On the south side it implements the interfaces to the NUBOMEDIA media elements.

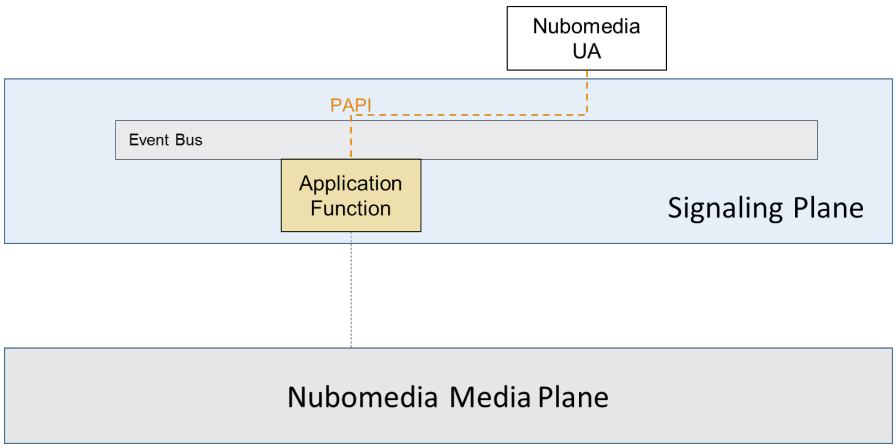


Figure 3 Application Function inside the Signaling Plane

6.2.1 Current Application Function

The current Application Function of Release 3 allows NUBOMEDIA clients only static execution of a service. This specific service allows two applications scenarios:

Video Call with augmented reality and recording

For this application service the client is connected via WebSockets and uses the session protocol to create a video call with another user. The application function creates a third party call with both users and the Kurento media server using the Kurento API. When the sessions are established the application function uses the Kurento API to record the video streams of both participants.

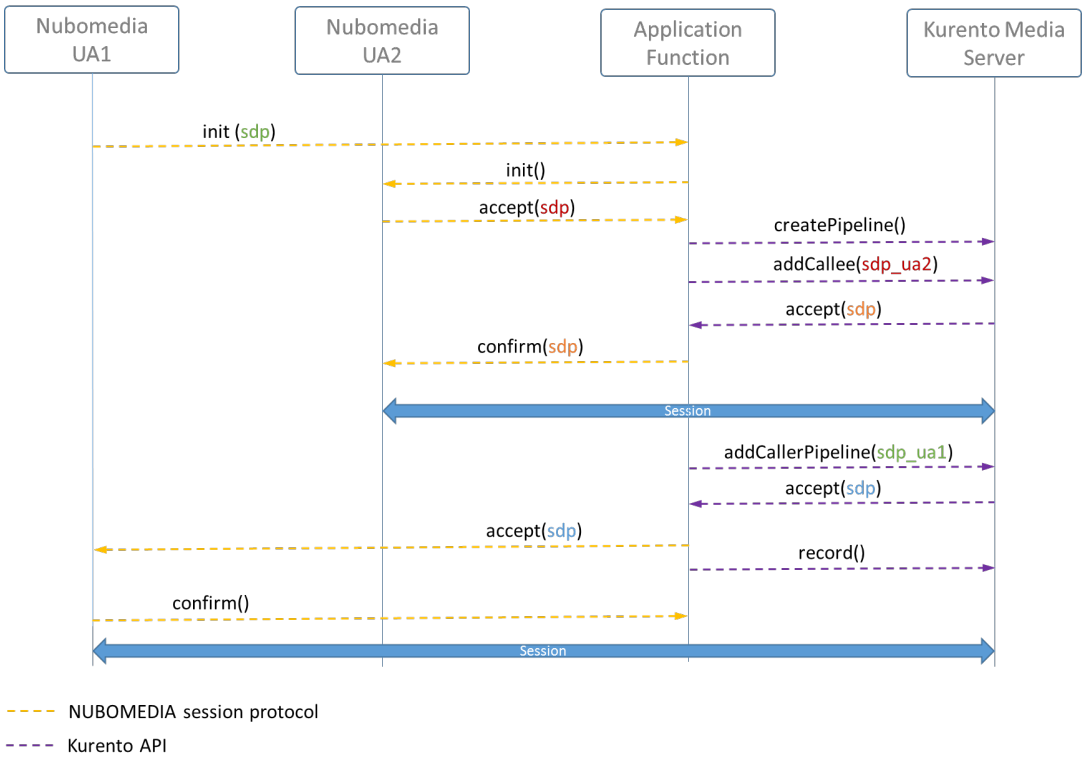


Figure 4 Video call with AR and recording

Video playback of a previous recorded video call

The second service of the application function is to start a playback of the previous recorded video call. In this case there is only one client that creates a video call with the application function in order to receive a video stream of the previously recorded file.

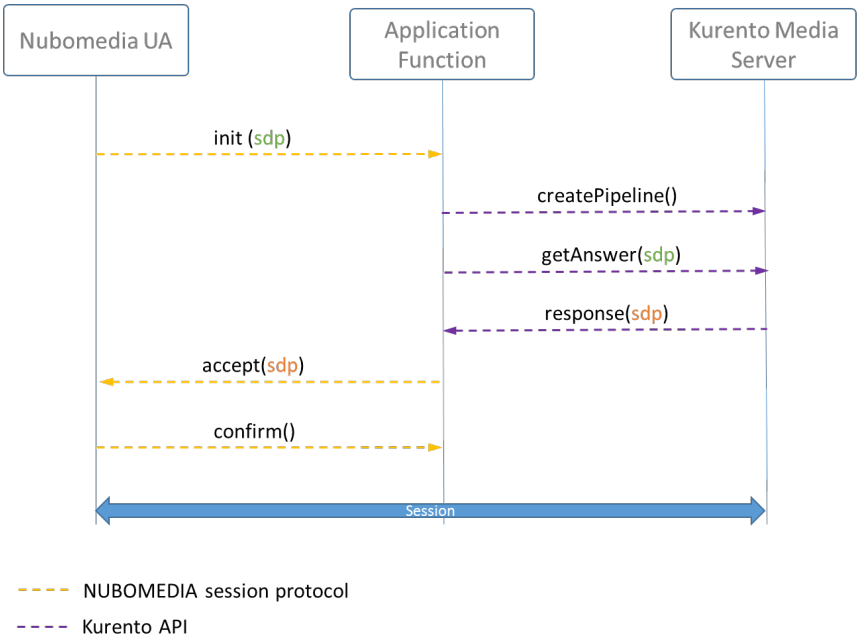


Figure 5 Play Video

The architecture of this version of the Application Function complies with Figure 3 with only one Application Function module.

6.2.2 Future Application Function

The envisioned version of the Application Function will be designed to support dynamic application services. This means NUBOMEDIA UAs can dynamically discover available NUBOMEDIA applications. Also the Application Function itself will be highly modular and scalable as illustrated in Figure 4.

As it can be seen in Figure 6, the Application Function consists of multiple modules. There are two types of Application Function modules:

- Application Service Discovery
  The discovery module provides an API (which may be part of the NUBO-PAPI) to NUBOMEDIA clients in order to discover which application services (represented by the service modules) are currently available on the NUBOMEDIA platform. This information might also contain information about how to interface with the specific service modules, since different application services will have different input and output parameters.

- Service Module
  A service module implements a specific application service (e.g. plays a recorded video). They register and unregister their service information at the discovery module in order to signal their availability for client service requests.

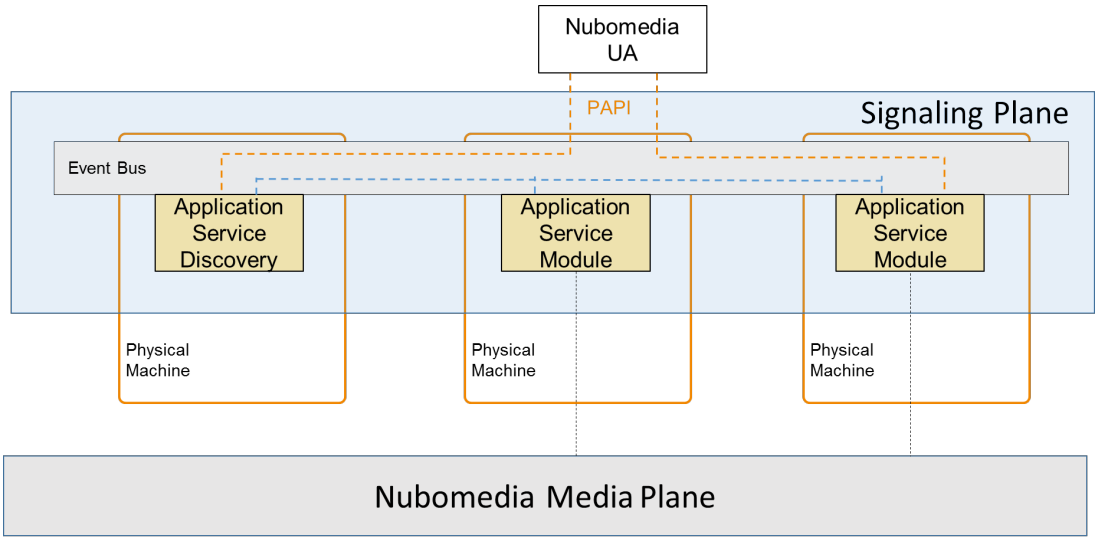


Figure 6 Modular Application Function

Figure 6 shows that Application Function modules have two different interfaces towards the Event Bus. The orange dotted lines are the PAPI interfaces which are used by NUBOMEDIA clients to use the services of the specific module. The blue dotted lines represent the interface for the service registration between the discovery module and the service modules.
Figure 7 shows how the application services can be discovered by the NUBOMEDIA clients. First the UA asks the discovery module for the available services and with the response the UA is able to create a media session with a service module, in this case with Service Module A.

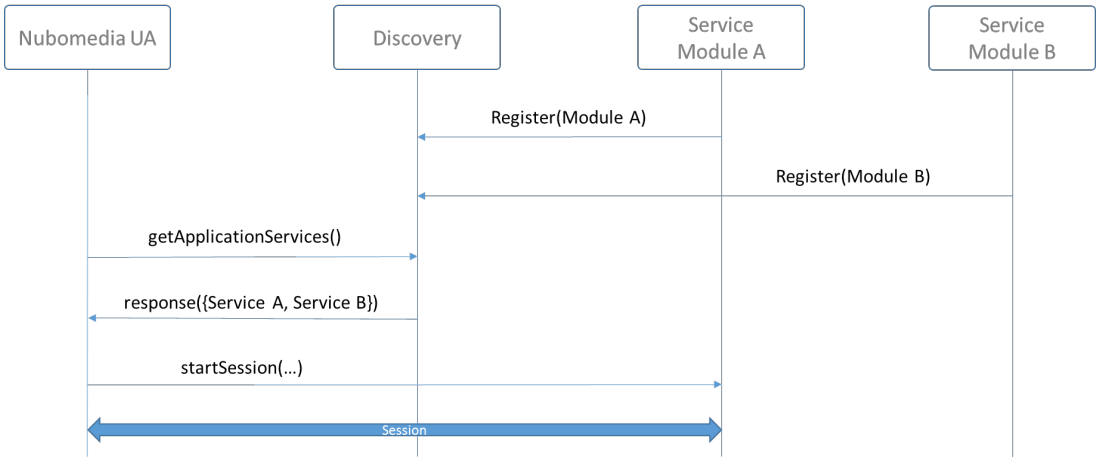


Figure 7 Application Service Discovery

This design of the Application Function allows a very flexible way of extending the platform: the Application Function can easily be extended with new application logic and the service modules can easily be deployed on runtime. NUBOMEDIA UAs can subscribe for an event and will be notified when new application service modules become available or unavailable.

Figure 6 also shows that the Application Function modules (like all modules connected to the event bus) may (but not must) be deployed on (physically) distributed machines. This allows also easy scaling of the Application Function e.g. for robustness or load balancing reasons.

7 Setup and Installation

7.1 Requirements

The signaling plane requires the following software components for the setup and installation process:

- UNIX-like OS (so far only Linux is tested)
- Java 7 JDK
- Git
- Python 2.7 or 3.x

7.2 Installation

First of all the sources of the signaling plane need to be downloaded from the git repository:

```
git clone http://80.96.122.50/frank.schulze/signaling-plane.git
```

As second step the sources of the Signaling Plane modules need to be compiled. The build process uses gradle as build tool. Run the script `compile_modules` to build all the modules.

7.3 Configuration

The Signaling Plane uses a JSON configuration file `nubomedia_signaling_plane.json`. This file contains the modules to be run and their configuration. The modules to be started are listed in a JSON array with the key "modules". In order to not start a module, the module just needs to be removed from that JSON array.

7.3.1 Cluster Configuration

This step is only necessary if OpenXSP instances should be run in clustering mode. This allows for connecting the event busses of the OpenXSP instances into a single transparent event bus and to do load balancing between the OpenXSP instances provided by the underlying Vertx / Hazelcast system.
The clustering can be configured in the file $OPENXSP_HOME/vertx/conf/cluster.xml file. There are basically two modes how the instances discover each other:

- IP broadcasting
- TCP-IP connections to one or more central nodes

The part of the configuration file to configure the basic clustering over multicast might look like this [8]:

```
…
  <group>
    <name>CLUSTER_NAME</name>
    <password>CLUSTER_PASSWORD</password>
  </group>

  <instanceName>INSTANCE_NAME</instanceName>

  <network>
    <port auto-increment="true">PORT</port>
    <join>
      <multicast enabled="true">
        <multicast-group>MULTICAST_IP</multicast-group>
        <multicast-port>MULTICAST_PORT</multicast-port>
      </multicast>
    </join>
  </network>
…
```

With the following parameters:

- CLUSTER_NAME
  Name of the cluster group. All nodes must share this name in order to be considered part of the group.
- CLUSTER_PASSWORD
  Password for the cluster group.
- INSTANCE_NAME
  Used to distinguish from multiple Hazelcast instances in the same JVM, if present. Typically, this setting will not need to be altered.
- PORT
  Port that Hazelcast uses.
- MULTICAST_IP
  Address for the multicast server. Typically this setting will not need to be changed.
- MULTICAST_PORT
  Port on which the multicast server operated. Typically this setting will not need to be changed.

If the setup does not support multicast, the configuration file may look like this:

```
…
<group>
    <name>CLUSTER_NAME</name>
    <password>CLUSTER_PASSWORD</password>
</group>

  <instanceName>INSTANCE_NAME</instanceName>

  <network>
    <port auto-increment="true">PORT</port>
    <join>
      <multicast enabled="false">
        <multicast-group>MULTICAST_IP</multicast-group>
        <multicast-port>MULTICAST_PORT</multicast-port>
      </multicast>
      <tcp-ip enabled="true">
        <interface>IP1</interface>
        <interface>IP2</interface>
      </tcp-ip>

    </join>
  </network>
…
```

… where IP1 and IP2 are individual IP addresses of the nodes in the cluster.

These are the basic configuration parameters for clustering which should suffice for most use cases.

7.3.2 IMS Connector Module

The IMS connector module needs to be configured with the local IP address and the port where it listens for SIP requests. Furthermore the IMS parameters need to be configured. The configuration of the IMS Connector module looks like this:

```
{
        "name": "org.openxsp~java_sip~0.1",
        "config": {
                "bind": "localhost",
                "port": "5060",
                "domain": "kamailio-ims.org",
                "ims_scscf": "13680",
                "ims_ip": "10.147.66.119",
                "ims_pcscf": "4060",
```

```
                "ims_icscf": "5060",
        }
}
```

The configuration parameters are explained as follows:

- `bind`
  The IP address which the module should bind to
- `port`
  The (local) port which the module uses to listen for SIP messages
- `domain`
  The IMS domain which the
- `ims_ip`
  The IP address of where the IMS components are running
- `ims_scscf`
  The port number where the S-CSCF of the IMS listens for SIP requests
- `ims_pcscf`
  The port where the P-CSCF of the IMS listens for SIP requests
- `ims_icscf`
  The port number where the I-CSCF of the IMS listens for SIP requests

7.3.3 Application Function Module

The application function module is called kurento_client. It needs to be configured with the IP address of the Kurento media server:

```
{
        "name": "org.openxsp~kurento_client~0.1",
        "config": {
                "kurento_address": "ws://192.168.149.134:8888/kurento",
                "player_event": "eu.nubomedia.af.kurento.play",
                "call_event": "eu.nubomedia.af.kurento.call.ar,"
                "service_registry_event": "eu.nubomedia.af.service_registry",
                "user_registry_event": "eu.nubomedia.af.user_registry"
        }
}
```

Configuration parameter:

- `kurento_address`
  This parameter points to a URI where a Kurento media server is listening for service requests.
- `player_event`
  This is the event name which a client uses to create a media session to play a recorded file. The event name probably won't change very often, but is has to be aligned with the event address that the client uses. That's why it's configurable.
- `call_event`
  This is the event name which a client uses to create a media session to create a call with another client enriched with augmented reality. The event name probably won't change very often, but is has to be aligned with the event address that the client uses. That's why it's configurable.
- `service_registry_event`
  This configuration parameter is not yet used. It will be used in later versions of the application function for the application service discovery as described in section 5.2.2.
- `user_registry_event`
  This configuration parameter is not yet used. It will be used in later versions of the application function to find out which to which event a specific user is listening.

7.4 Running the Signaling Plane

In order to run the modules, run the script "`nubomedia_signaling_plane`"

7.5 Summary

To summarize this section the following commands need to be executed to install and start the NUBOMEDIA Signaling Plane:

```
echo installing signaling plane
sudo apt-get install java7-jdk git python3.2
git clone http://80.96.122.50/frank.schulze/signaling-plane.git
echo compiling sources
cd signaling-plane
./compile_modules
echo TODO: configure signaling plane in nubomedia_signaling_plane.json
echo TODO: configure clustering if needed
echo starting signaling plane
./nubomedia_signaling_plane
```

8 Developing Application Modules

This section describes how extensions of the Signaling Plane in the form of new Modules can be developed on top of the OpenXSP platform. The aim of the OpenXSP platform is to provide a framework for developing loosely coupled and cloud enabled network service applications.

OpenXSP itself is built upon the vertx.io framework. Vertx.io - and thus OpenXSP - is a polyglot framework, meaning that applications can be written in a number of languages. Currently, OpenXSP supports Java and Python. However, vertx.io supports many more languages and supporting them in OpenXSP is just a matter of providing some library bindings.

For build management (both for the framework itself as well as any applications developed for it) the gradle build automation tool is used.

Before actually start writing modules, it is recommended to read the basic mechanisms of the OpenXSP platform discussed in chapter 4.

8.1 Creating new Modules

The unit of packaging for OpenXSP applications is called a "Module". A module can contain any number of (including zero) verticles and can depend on other modules (and their verticles) in turn. Creating a module with no Verticles makes sense to provide only library support for other modules.

8.1.1 Naming Scheme

Modules are identified by three components:

- the module owner
- the module's name
- the module's version

OpenXSP imposes little restrictions on either of these but it is sensible to stick to the following: the module owner should resemble a Java package name. Preferably the name of the package of the modules's source code (if it is implemented in Java)
Together, these components form a modules fully qualified name. For this matter, they are joined in a single string with a tilde (~) in between. Consider a module with the following:

```
owner: com.acme
name: myModule
version: 1.0
```

The fully qualified name of this module would be `com.acme~myModule~0.1`.

8.1.2 Module Skeleton

To start writing modules, the module skeleton that is provided by OpenXSP can be used. This provides a standard module layout as well as some infrastructure and build scripts. To create an empty project the initProject command of the OpenXSP CLI can be used:

```
openxsp initProject myProject
```

Optionally the --owner flag can be used to set the module owner. If it is left blank, the owner will be `org.openxsp`:

```
openxsp initProject myProject --owner com.acme
```

8.1.3 Module Structure

When using the default layout, a module directory has the following contents:

- `mod.json`
  Module descriptor
- `mods`
  External modules go here. Note that this folder only exists when dependencies are encountered.
- `src/main/java`
  Java sources go here
- `src/main/resources`
  Everything in here goes into the module distribution as it is. This is also the place to put source files to be distributed (e.g. python)

Note that in the above list, only the module descriptor (the mod.json file) is mandatory. All other entries exist because they are part of the standard module layout. It is possible to structure the module differently, but then you are on your own and much of this guide will not apply.

8.1.4 The Module Descriptor

Modules are described by a descriptor file: mod.json. As the name implies, it contains JSON data. A minimal descriptor looks like this:

```
{
  "owner": "org.acmecorp",
  "name": "myFancyShmancyModule",
  "version": "0.1"
}
```

The following fields are recognized in a module descriptor:

- `owner`
  The module's owner.
- `name`
  The module's name.
- `version`
  The module's version.
- `description`
  Text description of the module.
- `keywords`
  JSON array of strings representing keywords that describe the module.
- `developers`
  JSON array of strings representing any other developers of the module.
- `main`
  Indicates the startup routine for this module. For `example main.py` or `com.acme.MyPackage.MyClass`.
- `worker`
  Indicates that this is a worker module. Worker modules run using a thread from the background pool and are allowed to perform blocking operations. Standard verticles run using an event loop thread and cannot block. By default, a module is not a worker module.
- `multi-threaded`
  If the module is a worker then setting this to true makes the worker multi-

threaded, i.e. the code in the module can be executed concurrently by different worker threads. This can be useful when writing modules that wrap things like JDBC connection pools. Use this with caution. Default is false.

- `includes`
  Additional module dependencies as a comma-separated string of fully qualified module names. A module can include zero or more other modules. Including a module means the included modules classloaders are added as parents of the including module class loader. This means classes and resources from other modules can be accessed as if they were available in the including module. In some ways it's similar to importing packages in Java.
- `repositories`
  A list of directories that are searched for dependency modules. Note that OpenXSP's default module folder (${OPENXSP_HOME}/modules) will always be searched.

Thus, a more comprehensive module descriptor might look like the following:

```
{
  "owner": "org.acmecorp",
  "name": "myFancyShmancyModule",
  "version": "0.1",
  "includes":
"com.acme~anotherModule~0.1,org.openxsp~yetAnotherModule~1.5",
  "main": "main.py",
  "repositories": [
    "/my/favourite/module/folder",
    "/another/module/folder"
  ],
  "worker": false
}
```

8.1.5 Using other Modules from a Module

If a module ever refers to another module, e.g. because it wants to use resources from it or run the module (see below) that module must be specified in the `includes` section of the depending module's mod.json. Additionally, the module must be linked into the depending module's directory structure. That happens automatically if the child module can be found in one of the configured repositories (the recommended approach). If the latter is not the case, the `openxsp link` command can be used to manually create a module link.

In order to use the `openxsp link` command change to the parent module's directory (the directory of the module that depends on another module) and run the following command:

```
openxsp link <path_to_child_module>
```

8.1.6 Using Libraries in Modules

If your module directly uses other jar or zip files these should be placed in a lib directory which in the standard module layout resides under `src/main/resources`. Any jar or zip files in the lib directory will be added to the module classpath.
If a particular resource (i.e. jar or zip file) is used by more than one module, it makes sense to put those resources in a module of their own, and then declare that other modules includes them, instead of copying the same resources into every module that needs them. Doing this adds the classloader of the included module as a parent to the classloader of the including module, in effect meaning you can reference the resources in the included modules if they were present in the including module.

This is done by specifying an `includes` field in the parent module's descriptor. If the module cannot be found in one of the configured repositories, it must also be linked via the `openxsp link` command.
Modules that only contain resources for re-use are called non-runnable modules and they don't require a main field in mod.json. It's also possible to include the resources of runnable modules into another module.

8.1.7 Compiling Modules

In order to compile the source code of a module, the gradle build and dependency management tool is used. Simply go into the root folder of the module and execute the command `./gradlew` to compile the module.

8.1.8 Running Modules

Note that only runnable module - i.e. those that specify main in their mod.json can be run.
To simply run a *single* module, change to its directory and run the following command:

```
openxsp runmod
```

This will cause openxsp to attempt to resolve any module dependencies using the configured repositories and then invoke the startup routine specified in main.

8.1.9 Passing Configuration

Optionally, a configuration file can be passed to the `runmod` command. This configuration is always a JSON file that contains a single JSON object with arbitrary data. To pass configuration, use the `--conf` option to `openxsp runmod`:

```
openxsp runmod --conf config.json
```

In Java, that configuration is then available inside your verticle by calling the `config()` method on the container member variable of the verticle:

```
JsonObject config = container.config();
System.out.println("Config is " + config);
```

In Python, that configuration is available to the verticle using the `openxsp.config()` method. For example:

```
import openxsp
config = openxsp.config
print ("Config is: ", config)
```

8.1.10 Running multiple Modules as an App

Multiple modules (but also single modules) can be contained in what is called an App. An app is simply a JSON descriptor file that resides in its own directory. This descriptor file specifies a number of modules to be run along with their configuration.
Such an app can be run with the `openxsp runapp` command:

```
openxsp runApp app.json
```

The following fields are recognized in an app descriptor:

- `owner`
  The app's owner.
- `name`
  The app's name.
- `version`
  The app's version.

- `description`
  Text description of the app.
- `keywords`
  JSON array of strings representing keywords that describe the app.
- `developers`
  JSON array of strings representing any other developers of the app.
- `modules`
  JSON array of module entries. Each module entry describes one module to be started.

The following fields are recognized in a module entry:

- `name`
  The fully qualified name of the module to be run (mandatory).
- `instances`
  Number of instances to run. Use with caution.
- `config`
  The module's config.

An example of such a descriptor is shown here:

```
{
    "owner": "org.openmtc",
    "version": "0.1",
    "name": "test app",
    "repositories": [
        "examples"
    ],
    "modules": [
        {
                "name": "org.example~new_python_example~0.1",
                "instances": 1,
                "config": {
                        "message": "World"
                }
        },
        {
                "name": "org.example~another-module~0.1",
                "instances": 1,
                "config": {
                        "foo": "bar"
                }
        }
    ]
}
```

8.1.11 IDE Support

Via gradle it is possible to create an eclipse project file. After this the module can be imported as project into eclipse or also IntelliJ IDEA. Therefore the following command needs to be executed in the root folder of the module:

```
./gradlew eclipse
```

8.1.12 Summary

Summarizing this chapter the following steps need to be executed to create a simple module:

1. Check out the source code
   ```
   git clone http://80.96.122.50/frank.schulze/signaling-plane.git
   cd signaling-plane
   ```

2. Create a new module
   ```
   openxsp initProject myProject
   cd modules/myProject
   ```

3. Write a Verticle
   Create e.g. a Java class that extends `org.openxsp.java.Verticle` in the `src/java/main` folder. Examples can be found in the next chapter.

4. Configure the Verticle in the module descriptor ./`mod.json`

5. Compile the module
   ```
   ./gradlew
   ```

6. Run the module
   ```
   openxsp runmod
   ```

8.2 Network Communication

With OpenXSP it is possible to create applications that communicate via different signaling technologies. This can be applications based on HTTP, WebSockets, SockJS, SIP or plain TCP/UDP servers. For most of these technologies, OpenXSP relies on the underlying Vert.x APIs and therefore the Vert.x documentation [7] can be used for the development of such applications.

There is also a SIP library module available in the `modules` folder called `lib-sip` which can be used to create SIP server and client endpoints. The current status of the implementation provides only a very low level API for application developer. For later releases it is planned to create an abstraction layer so it is more convenient to develop SIP signaling modules.

8.3 Event Bus Protocol

An implementation of the Event Bus Protocol as described in chapter 5 is available as library module. It can be found in the `modules` folder at `lib-sessioncontrol`.

Source Code Examples

The following snippets show how the implementation of the event bus protocol can be used.
Receiving events:

```
public void handle(Message msg) {
        // dispatch message
        JsonObject msgJsonBody = (JsonObject) msg.body();

        SessionControlMessgeImpl impl = new SessionControlMessgeImpl();
        SessionControlMessage m = impl.parse(msgJsonBody);

        switch (scm.getMethod()) {
                case INIT:
                        //handle session initiation
                        SessionControlCreateSession cMsg= (SessionControlCreateSession) m;
                        String applicationSpecificContent = cMsg.getContent();
                        //TODO do something with the content …

                case ACCEPT:
                        SessionControlAcceptSession acc = (SessionControlAcceptSession) m;
                        //handle session accepted …

                case CONFIRM:
                        SessionControlConfirmSessionEstablishment confirm =
                                        (SessionControlConfirmSessionEstablishment) m;
                        //handle session confirmed
```

```
case CANCEL:
    SessionControlCancelSession cancel = (SessionControlCancelSession) m;
        //handle session canceled …

case END:
        SessionControlEndSession end = (SessionControlEndSession) m;
        //handle session ended

case UPDATE:
        SessionControlUpdateSession u = (SessionControlUpdateSession)m;
        //handle session update
```

Sending session events:

```
//send a create session request
SessionControlCreateSessionImpl c = new SessionControlCreateSessionImpl();
c.setContent("some content");
c.setContentType("plain/text");
c.setFrom("me");
c.setTo("receiver_name");
c.setAddress("my_callback_adress_for_session_accept_response_or_other_response");
c.setSessionID("my id for the session");

String remoteEvent = "event_address_to_where_the_message_shoul_be_sent";

EventBus eb = openxsp.eventBus();
eb.invoke(remoteEvent, "rpc_action", c.create(), new RPCResultHandler() {
      …
      });

//send session accept
SessionControlAcceptSessionImpl a = new SessionControlAcceptSessionImpl();
a.setContent("content");
a.setContentType("plain/text");
a.setSessionID(getSessionIdFromCreateRequest());
a.setAddress("my_callback_adress_for_session_confirm_response_or_other_responses");
String remoteEvent = getSessionRemoteEventFromCreateRequest();

eb.invoke(remoteEvent, "rpc_action", a.create(), new RPCResultHandler() { …});


//sending session confirm response
SessionControlConfirmSessionEstablishmentImpl c = new
                                    SessionControlConfirmSessionEstablishmentImpl();
c.setContent("content_to accept_session");
c.setContentType("text/plain");
c.setSessionID(getsessionid());
c.setAddress("my_callback_adress_for_session_events");

String remoteEvent = getSessionRemoteEventFromAcceptResponse();

eb.invoke(remoteEvent, "rpc_action", c.create(), new RPCResultHandler() {…});
```

…

An example where the session control protocol has been used is the application function (the `kurento_client` module in the `modules` folder; in the `eu.nubomedia.af.kurento.net.vertx.VertxHandler` class).

8.4 Example Applications

This section will introduce two example applications. These very simple applications consist of a receiver application and a sender application. The receiver application will subscribe for an event by registering a handler at the event bus. The second application will publish some information for the event that the receiver application has registered a receiver for.

8.4.1 Receiver Module

```
public class Receiver extends Verticle {
	public static final String EVENT = "org.openxsp.test";
	@Override
	public void start() {
		System.out.println("Registering event handler for "+EVENT);
		EventBus eb = openxsp.eventBus();
		//Register a handler for my event
		eb.registerHandler(EVENT, new Handler<Message>() {
			@Override
			public void handle(Message msg) {
				System.out.println("Received event and replying ...");
				JsonObject res = new JsonObject();
				res.putString("status", "ok");
				msg.reply(res);
				Object data = msg.body();
				//TODO do something with the data …
			}
		});
	}
}
```

This code snippet shows the main part of the receiver application. In this example it consists of a Java class. This java class extends a `Verticle` class. A Verticle is the main unit of execution in OpenXSP. Each module (except library modules) has a main Verticle that is started upon starting of the Module. A Module can also have further Verticles. These verticals can be started by any other Verticle, e.g. the main Verticle.
A Java class that extends the Verticle class must implement the `start()`-Method. This method is called by the OpenXSP framework when the Module is starting. It can be used to register handler for specific events (like in this example) or to do some other initialization procedures.
In this case, the Verticle obtains an `EventBus` object using the `OpenXSP` object that has been inherited from the Verticle class that the Sender class is extending.
The EventBus is now used to register a Handler for a specific event. Once something is published for this event, the `handle(Message)` method of the handler is invoked here the sender can now reply with a status message and process the received data.

8.4.2 Sender Module

```
public class Sender extends Verticle {
	public static final String EVENT = "org.openxsp.test";

	@Override
	public void start() {
		System.out.println("Starting Sender Example module");

		System.out.println("Type a message to send:");
		Scanner reader = new Scanner(System.in);
		String message = reader.next();
		reader.close();
		sendEvent(message);
	}

	private void sendEvent(String message){
		JsonObject data = new JsonObject();
		data.putString("message", message);
		System.out.println("Sending message "+data+" to "+EVENT);
```

```
                openxsp.eventBus().invoke(EVENT, " ", data, new RPCResultHandler() {
                        @Override
                        public void handle(RPCResult res) {
                                if (res.succeeded()) {
                                        System.out.println("Successful reply received");
                                }
                                if (res.failed()) {
                                        System.out.println("Unsuccessful reply received");
                                }
                        }
                });
        }
}
```

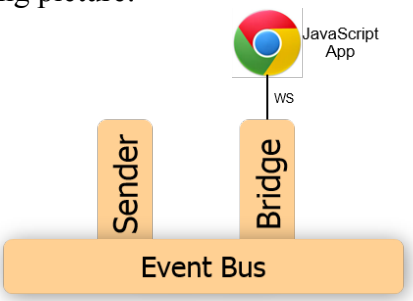
In this example, a Verticle is created that publishes some data on the event bus which is read as console input. Note that this is a very special use case, but this example shall only show a simple use case.
Like in the previous use case all the work is done in the `start()`-method. Again the `EventBus`-object is used to interact with the event bus of OpenXSP. In this case the `invoke`-method is used to publish the data for a specific event. The `invoke`-method sends the data in a peer to peer way to a receiver, even if there are more received subscribed to this event. This means the event is only received by one handler. The data can also be broadcasted to all receiver that have registered for the event using the `send()` method of the `EventBus` object.
The last parameter of the invoke-method an `RPCResultHandler` object. This handler handles the replies that have been send by the receiving handler using the Message-object and the `reply` method.
In this case the module does some blocking I/O while it waits for the user input. This means the module should be run as worker module to not block the complete event bus. Otherwise it could not handle events before the user has entered something on the console.

8.4.3 Alternative Receiver Module

As has been mentioned, OpenXSP is a polyglot platform. This is an alternative receiver application that does basically the same as the first receiver application, but which is written in JavaScript. JavaScript is not directly supported by OpenXSP, but since OpenXSP is based on vertx.io modules can also be written in JavaScript.
A JavaScript Module consists of two parts:

- JavaScript application
  The JavaScript application that implements the logic that the module provides
- Bridge
  A component that connects the JavaScript application with the event bus

This is shown in the following picture:



8.4.3.1 Bridge

The Bridge connects the JavaScript application with the event bus and has basically two tasks:

- Provide WebSocket for the JavaScript application
  The JavaScript will need to connect to the Event Bus. The method of choice is here to open a WebSocket connection that is used to publish and/or receive events. A WebSocket is a permanent bidirectional TCP connection between a client application and a server. The server in this case is the bridge.
- Access control
  By using the WebSocket, the JavaScript application is able to access directly the event bus. This is a security issue, since any JavaScript client application could listen and write on the event bus. Therefore a filter mechanism is introduced that allows the JavaScript application to only send and receive specific events.

Below is an example of a bridge written in Java. It could also be written in any other of the supported languages.

```
public class BridgeServer extends Verticle {
        public void start() {
                System.out.println("Starting Websocket server");
                HttpServer server = vertx.createHttpServer();
                server.requestHandler(new Handler<HttpServerRequest>() {
                    public void handle(HttpServerRequest req) {
                        String file = "";
                        if (req.path().equals("/")) {
                          file = "index.html";
                        } else if (!req.path().contains("..")) {
                          file = req.path();
                        }
                        System.out.println("Returning file "+file);
                        req.response().sendFile("/path/to/webapp/" + file);
                    }
                  });
                JsonObject config = new JsonObject().putString("prefix", "/eventbus");
                //create access rights filter - in this case empty = no restrictions
                JsonArray noPermitted = new JsonArray();
                noPermitted.add(new JsonObject());
                openxsp.createSockJSServer(server).bridge(config, noPermitted, noPermitted);
                server.listen(8080);
        }
}
```

Again, all the necessary logic is done in the `start`-method of the Verticle. An HTTP server is created and a Handler is assigned to this server that serves HTTP requests with a file. This file contains the JavaScript application. As next step the server socket is created given the access rights filter - one for incoming events and one for outgoing events. The filter are described as JSON objects, containing regular expressions that whitelist the allowed events to be published or subscribed.
Finally the server is started on a specified TCP port.

8.4.3.2 JavaScript Application

The jacaScript application does basically the same as the Receiver written in Java. It retrieves the event bus variable to register an event handler for a specific event. Once this event happens, a callback method is invoked that replies a status messages and writes the received data into a log screen.

```
<script>
    function log(msg) {
        console.log(msg);
        $("#log").append('<div>' + msg + '</div>');
    }
    var eb = new vertx.EventBus('http://localhost:8080/eventbus');
    var EVENT = "org.openxsp.test";
    log ("Running...");
    eb.onopen = function() {
      log("Registering at address " + EVENT);
      eb.registerHandler(EVENT, function(message, reply) {
        log('received a message: ' + JSON.stringify(message));
        var replyMsg = {status: "OK", msg: "pong_from_" + EVENT};
               log("replying with: " + JSON.stringify(replyMsg));
```

```
                reply(replyMsg);
                log("reply sent");
        });
    }
</script>
```

For a more exhaustive introduction in how to write JavaScript application please visit also the official API and tutorial pages of Vert.x.

8.4.4 Example Source Code

The source code of the examples can be found on git:

```
git clone http://80.96.122.50/frank.schulze/signaling-plane.git
cd signaling-plane/modules
```

To run the modules either run the `nubomedia_signaling_plane_examples` script in the root folder or start each module separately with the command `openxsp runmod -C` from inside the root folder of the module.
Before the modules can be run they have to be compiled. To do this, execute the command `./gradlew` in each module folder.

8.5 Module Repositories

Modules can be published into so called repositories. Before developing own modules, it is recommended to search in existing repositories for modules implementing the desired functionality. There is a big community developing modules for Vert.x. Vert.x modules are 100% compatible with OpenXSP. There are a couple of public repositories with hundreds of modules, as for example:

- http://modulereg.vertx.io/
- http://mvnrepository.com/artifact/io.vertx
- https://github.com/vert-x

References

[1] Vert.x, http://vertx.io/
[2] NodeJS, http://nodejs.org
[3] gevent, http://www.gevent.org
[4] http://callbackhell.com/
[5] "Node.js and The Case of the Blocked Event Loop", http://zef.me/4561/node-js-and-the-case-of-the-blocked-event-loop
[6] Vert.x JavaScript API Manual, http://vertx.io/core_manual_js.html
[7] Vert.x Documentation, http://vertx.io/docs.html
[8] http://suite.opengeo.org/4.1/sysadmin/clustering/params.html#hazelcast-xml