API for Shongo

Petr Holub, Jan Růžička, Miloš Liška, Martin Šrom, Ondřej Pavelka, Ondřej Bouda

© CESNET z. s. p. o.
2012

Contents

Chapter 1

Use Cases

1.1 Common

UC-1 (com:identification) Entity identification

Each entity in Shongo (e.g., resource, or reservation) is identified by an unique identifier. The identifier follows the URI standard [**?**]:

`shongo:<type>:<domain>(.<subdomain>)*:<uuid>`

The `<type>` component represents **`type`** of an entity (e.g., `resource`, or `reservation`). The `<domain>` component represents the full name of a **`domain`** to which the entity belongs (or where it was created in case of reservation). Each **`domain`** will run it's own controller platform. The `<uuid>` component represents represents UUID [2] which is generated automatically and it provides uniqueness in distributed systems without need for central coordination.

1.2 Resources

UC-2 (res:types) Types of resources

Basic resource types include the following:

A managed endpoint This is an endpoint, that is managed by Shongo – the endpoint is both managed (calls are automatically dialed when involved in reservation, directory is updated, etc.) and monitored (availability and status).

A unmanaged endpoint This is an endpoint, which is not available for Shongo management for either technical or administrative reason. It may be, e.g., a software H.323 client or web browser acting as a Adobe Connect client. Its specification by a user (e.g., providing attributes like H.323 number or H.323 ID), however, allows for specific adjustments during implementation of the reservation – e.g., monitoring of participants in the calls and allowing only participants calling from specific H.323 number or ID.

A managed infrastructure element This is one of the infrastructure resources, that is managed, monitored and typically also scheduled by the Shongo. It includes things such as H.323 MCUs, H.323 gatekeepers, Adobe Connect servers, recording servers, streaming servers, and various types of gateways and translators.

A virtual room A virtual room is a private compartment on a specific multi-point infrastructure element. Typically, this is comes as a product of a scheduling process. Virtual rooms are often not licensed, only their participants are (this is a concurrent user license model). However, other models may also exists and this is abstraction allows for them.

A license This is typically the limiting factor of infrastructure elements in a concurrent user licensing model. Utilization of the licenses is scheduled by Shongo, while some licenses may be put aside as a part of permanent reservations by a resource owner (see UC-10).

A physical room This a representation of a physical meeting room and Shongo thus allows for reserving physical rooms. Its representation among the resources enables also more advanced uses: a physical room may contain multiple videoconferencing devices and reserving a room also means that the those devices become unavailable for other reservations than the one which contains the physical room.

A specific identifier A user may reserve a specific identifier, typically Adobe Connect URL, H.323 number, or streaming server URL. This allows for reuse of such an identifier in irregularly recurring and *ad hoc* events.

UC-3 (res:management) Management of resources

The resource owner should be able to create new resources that will be managed by Shongo. Owner should be able to modify the managed resource parameters and also should be able to delete the managed resource.

UC-4 (res:identification) Resource identification

Each resource is identified by an unique identifier as defined in UC-1. The identifier will be assigned to the resource by it's owner when the resource is being created. The format is:

`shongo:resource:<domain>(.<subdomain>)*:<uuid>`

Each resource belongs to some main **`domain`** that runs controller platform. The domain can be structured to any number of **`<subdomains>`** by "`.`" sign. The <uuid> represents UUID [2] which is generated automatically and it provides uniqueness in distributed systems without need for central coordination. For unmanaged resources that don't belong to any domain there is a default `unmanaged` domain.

`TODO:` `Modify examples to use UUID`

Examples of managed resources:

- `urn:shongo:resource:cz.muni.fi:sitola.c90` – H.323 endpoint at C90 room
- `urn:shongo:resource:cz.cesnet:mirial.srom` – personal H.323 Mirial endpoint

Examples of unmanaged resources:

- `urn:shongo:resource:unmanaged:h323.<H.323 id or number>`
- `urn:shongo:resource:unmanaged:connect.<shibboleth identity>`

By a resource identifier, the user can lookup the resource type and all other attributes.

1.3 Reservations

UC-5 (rsv:specifications) Types of specifications

Specification of a resource, being object of a reservation, may be of the following types:

- a *fully-qualified explicit specification (FQESpec)* – specifies exactly one element; it ma refer to a specific device (e.g., H.323 endpoint, web browser as an endpoint for Adobe Connect), a specific server (e.g., a specific Adobe Connect server or H.323 MCU), a specific physical room, or a specific virtual room (e.g., a specific room running on specific H.323 MCU),
  FQESpec may be managed by Shongo or not; for resources that Shongo does not manage or knows about, i.e., unmanaged resources, the user needs to specify type of the resource (e.g., generic H.323 endpoint). The unmanaged resources should have some form of identification (e.g., H.323 number, H.323 ID, or Shibboleth identity for Adobe Connect) so that Shongo can verify if they are connected to the virtual room or not during the conference.
  Anonymous unmanaged resources may also be available (completely generic H.323 enpoint without a number or H.323 ID, or guest user in Adobe Connect) , but some functionality may not be available – when maximum room capacity is achieved (or exceeded), anonymous users not be allowed in (or even be disconnected in LIFO mode until maximum amount of participants is obeyed).
- a *partially-qualified explicit specification (PQESpec)* – specifies a class/type of a resource (e.g., H.323 endpoint) and it is up to the scheduling to find suitable one (combination of availability and access-level for given user),
- a *implicit specification (ISpec)* – the user does not specify such a resource, but the resource is needed to implement user's request (e.g., if user specifies Connect and H.323 endpoints, a gateway/connector is needed to implement the translation; if user specifies multiple H.323 endpoints beyond MCU-capability of each of them, some MCU is needs to be included).

Generally, Shongo should use the technology to limit number of participants in the rooms created based on the reservations—e.g., H.323 MCUs allow for setting an upper limit on number of participant in each room.

UC-6 (rsv:roles) User roles

Each reservation should have at least two types of possible user roles:

- *owner/administrator*, who can modify or even delete the reservation,
- *manager*, who can control the room (e.g., disconnect participants, mute participants, etc.),
- *participant*, who can only view the reservation including coordinates necessary for participation.

The roles can be delegated, which is important especially in case of owner/administrator: the original reservation creator can delegate this role to other users and any of them can the modify or delete the reservation.

UC-7 (rsv:identification) Reservation identification

Each reservation is identified by an unique identifier as defined in UC-1. The identifier is

assigned to reservation automatically by the controller platform. The format is:

`shongo:reservation:<domain>:<uuid>`

Each reservation is created in a **`domain`** that runs it's own controller platform. The <uuid> represents UUID [2] which is generated automatically and it provides uniqueness in distributed systems without need for central coordination.

UC-8 (rsv:reservation:one) One time reservation

Common type of reservation, where a user requests certain resources for limited time duration. Unlimited reservations are not assumed by this scenario (see UC-10).

Start time of a reservation may be any time in the future or *now*, which is also called *ad hoc* reservation.

Reserved resources may be given as FQESpec, PQESpec, or ISpec. FQESpec are either accepted or denied by the scheduler, while other types of the specifications are looked for their best match. PQESpec may include the following:

- user may request a general endpoint and Shongo should try to find the closest matching endpoint available to the user (e.g., user requests a H.323 endpoint for a conference since she has no personal endpoint, and she is assigned a room-based H.323 endpoint provided the room is available),

while examples of ISpec are as follows:

- amount of central resources (such as H.323 MCU ports or Connect licenses) based on specified number of (H.323/SIP or web-browser) participants,
- any interconnecting elements (e.g., gateways) to interconnect the endpoints specified by the user; if only part of the endpoints can be interconnected, the user should be notified what parts can be interconnected and what parts are disconnected.

Each reservation has to be given a unique identifier that is further used for any references to it. If the reservation is denied, reasons for denying should be communicated to the requester. In case that the reservation succeeds, all the users involved should be notified.

Each reservation has to include:

- unique identifier,
- timespan definition,
- requester's identifier,
- name,
- links to the resources involved, including specification of the amount of resources consumed,
- list of users involved.

Reservations may be compounded to form another reservation. This allows to reuse elements that are already reserved (e.g., a specified identifier or allocation of a physical room) to implement a larger reservation. As a part of the scheduling process, the scheduler has to check whether the reservation times and durations are compatible.

UC-9 (rsv:reservation:periodic) Periodic reservation

UC-8 extended with periodicity. Expressiveness of the periodicity language should be equivalent to cron plus start time, stop time or number of repetition, and explicit lists for recurring aperiodic requests.

UC-10 (rsv:reservation:permanent) Permanent reservation

This is specific type of reservation that can be only made by an owner of the resource as it permanently removes the reserved capacity from the dynamic Shongo scheduling.

Even permanent reservations must not threaten what has already been reserved for any user. In case of priority requests (see UC-11), Shongo must be able to migrate the reservation to other resources.

The difference between permanent and periodic reservation is that for permanent reservations is not applied the maximum future time as defined in UC-12. The permanent reservation also has bigger priority than periodic reservation (e.g., in scheduler input queue).

UC-11 (rsv:priority) Priority reservations

Priority reservations are only allowed by an owner of the resources and they may affect reservations already present on the resources. However, priority reservation should only be allowed if there is some other resource(s) (maybe even in another domain) that can take over the prior reservation. In case of reservation migration, all the involved users must be notified (see UC-17).

`TODO: We need to decide, whether to allow this or not.`

UC-12 (rsv:max-future) Maximum future time for reservations

Each resource owner should set a date/time limit in the future (e.g., 2 months), above which reservations are not allowed. That should be done for each owned resource. Whole reservation duration must fit in that limit. This limit ensures there is some time point in the future, where there are no reservations on the resource—e.g., for maintainance purposes, removal of the device, special events the device will be used for, etc.

UC-13 (rsv:lookup:time) Lookup available time

User may look up available time slots for given amount of requested resources, with either inter-domain negotiation turned off or on (i.e., tell the user when resources are available within the domain or when merging resources of all the domains).

UC-14 (rsv:list) List all the reservations

Some querying/filtering language needs to be supported to limit list to

- room types (H.323, SIP, Connect, etc.),
- equipment (be it class of equipment or a specific device).
- reservation owner(s),
- users involved (may be humans as well as resources, such as rooms with equipment) involved in the room as participants.

UC-15 (rsv:modify) Modification of a reservation

Any attribute of a reservation may be requested to change. The request may be accepted or denied by the scheduler. In case of the denial, reasons for denial should be communicated to the requester. If the modification succeeds, all the users involved should be notified.

UC-16 (rsv:release) Release/canceling of a reservation

All the users involved should be notified.

UC-17 (rsv:migration) Migration of a reservation

If the change is visible to the users (e.g., typically this would include change of the server/MCU the users connect to), all the users involved should be notified.

UC-18 (rsv:notification) Notification of participants

In case of making, modifying, or canceling a reservation, all the users involved should be notified, as specified in UC-8, UC-9, UC-10, UC-15, UC-16, and UC-17. By default, the users should be notified via email, but it would be interesting to provide also SMS notification service.

UC-19 (rsv:service-users) Reservations of rooms, public or semi-private endpoints, etc.

Each reservation may include endpoint resources (beyond human users with private endpoints—H.323/SIP/web), which represent entities such as rooms, non-personal endpoints, etc., that can be scheduled in a similar way to central resources.

This type of reservation may be either part of some infrastructure reservation (see UC-8, UC-9, UC-10) or standalone reservation (e.g., reservation of a meeting room with H.323 equipment to disable the room from scheduling for given time duration).

UC-20 (rsv:recording) Reservation of recording capacity

Usually part of some infrastructure reservation (see UC-8, UC-9, UC-10), but may be completely standalone in case that only recording server is used of the Shongo-managed infrastructure.

UC-21 (rsv:streaming) Reservation of streaming capacity

May part of some infrastructure reservation (see UC-8, UC-9, UC-10), but may be completely standalone in case that only streaming server is used of the Shongo-managed infrastructure.

1.4 Operations

UC-22 (ops:migration) Live migration of a virtual room

This use case is intended for migration due to planned server maintenance or unplanned server outage. Ideally, all the room settings and content should be transferred to the target room—but some content may be lost in case of unplanned server failure (namely content migration).

Being able to transfer room settings to another server in case of unplanned failure also requires that the settings needs to be stored in the Shongo middleware.

Clients should be automatically redirected to the new server, if technology permits, or at least notified of the migration (email, SMS—see UC-18).

Some functionality will be common UC-17.

1.4.1 Room Management

UC-23 (ops:room:shongo-options) Get room information on Shongo level

This information typically includes name, owner, date/periodicity, duration and type.

UC-24 (ops:room:users-list) List users

Each user should be given a unique identifier in the output list that can be used for further querying. It should also provide means to identify the same user (e.g., if the user disconnects–reconnects, it should contain a part that is common and that denotes the specific user and a part that is specific for the session, so that if the user is connected twice (one session is in timeout state and the other session has just been established), we can differentiate between the two sessions).

UC-25 (ops:room:user-info) Print detailed info about a user in a room

Print all the statistics we can get about a user participating in the room. It should contain technology agnostic part (e.g., when the user joined) and technology specific part (i.e., H.323 statistics, H.245/SIP capabilities negotiation info, H.239 content information, etc.).

TODO: Could the use case be more specific regarding the technology specific part? What does H.323 statistics and others look like? Should a class be defined for each such a technology-specific information?

UC-26 (ops:room:layout) Set room layout

Shongo should be able to set up global layout of a room and user-specific layout, if available through API of virtual room provider.

UC-27 (ops:room:user-disconnect) Disconnect a user

Immediate disconnection of a user.

UC-28 (ops:room:disable-user-content) Disable H.239 content from a specific user

Disable content the user to be H.239 content provider for the given room.

UC-29 (ops:room:specific-user-content) Enable H.239 content only from a specific user

Enable H.239 content only from the specific user, typically by disabling content from all other users. Normally, users may fight who is going to be the content provider.

UC-30 (ops:room:user-mute) Mute a user

Mutes user on the room level. Optionally if user's endpoint is also controlled by Shongo, it should provide means to mute the endpoint (which can be easily unmuted by the user).

UC-31 (ops:room:user-miclevel) Set microphone audio level for a user

Sets the audio from the user on the room level. Optionally, if user's endpoint is also controlled by Shongo, it should provide means to control mic level on the endpoint. In this case, audio should be normalized on the endpoint before doing modifications on room level (if the sound is too low or too high and distorted, it may not be corrected on the MCU).

UC-32 (ops:room:user-playlevel) Set playback audio level for a user

This functionality is typically available only when user's endpoint is also controlled by Shongo.

UC-33 (ops:room:user-video-off) Disable video of a user

UC-34 (ops:room:user-video-snap) Video snapshot for a user

If provided by the room provider (MCU, web conferencing, etc.), we should be able to get video snapshot of:

- video sent by the user,
- video received by the user.

UC-35 (ops:room:user-layout) Set layout specific for a user

UC-36 (ops:room:settings-down-up) Download and upload room settings

We should provide an API that allows for downloading settings of the room to the maximum extent possible, in order to back it up and reupload it later on. This is a convenient way to back up setting as well as to reset a newly created room (e.g., as a part of a new reservation) to old settings.

UC-37 (ops:room:content-down-up) Download and upload room content (if technology permits)

If technology and access policy permits, we should be able to download and upload content of the room (e.g., documents, notes, polls, etc.). See UC-36.

UC-38 (ops:room:room-techspec) Get/set technology-specific properties for a room

This may include specific attributes of the room (typically on room provider level), such as enabled codecs.

UC-39 (ops:room:user-techspec) Get/set technology-specific properties for a user

UC-40 (ops:recordings-management) Management of recording archives

It should be possible to work with the recorded video through Shongo, e.g., migrate it from a content server to a storage of a streaming server. Plus it should be possible for

owner/administrator or manager to access URLs of the recorded content to send them via email. Also, it should be possible to automatically notify all the (non-anonymous) participants about the recording via email.

1.5 Monitoring & Management

1.5.1 Shongo management and monitoring

UC-41 (mgmt:shng:list-agents) List of all the agents in the system

The listing API must include querying language that allows selection of only a subset based on similar properties like those defined in UC-14.

UC-42 (mgmt:shng:list-controllers) List primary and backup controllers

List all the controllers (primary and backup) for current domain.

UC-43 (mgmt:shng:list-domains) List domains

List of all other known domains including references to their domain controllers and state of connections to them.

1.5.2 Server management and monitoring

UC-44 (mgmt:srv:get-load) Get server load

The API should provide means to get load on the server machine, containing at least the following:

- CPU load
- memory load
- disk occupancy

Obviously, this information may or may not be available for specific device. In case that the information is not available, the API should report this in a consistent way (specific exception or unique return value).

UC-45 (mgmt:srv:schedule-downtime) Schedule server downtime

Downtime scheduling must include change/migration of all the reservations and live events influenced by the downtime. Conceptually, this is similar to permanent reservations a bit (UC-10)—the major difference is that during the downtime, the resource is not available to Shongo for management and this state is intentional. Downtime is also per-resource and does not have participants.

UC-46 (mgmt:export-stats) Export Shongo stats

Export reservation stats in some common format like CDR. `TODO: Specify in more detail - what stats?`

Chapter 2

Common Data Types and Object Classes

In this chapter we describe atomic types, enum types and object classes. In XML-RPC, every atomic type is converted to it's equivalent and each enum is converted to **`String`**. Object instances are converted to XML-RPC's **`struct`** and each non-empty object's attribute to XML-RPC **`member`** which consists of **`name`** and **`value`**. Every **`struct`** also includes one special `class` member which defines object class.

We use XML-RPC empty **`struct`** to represent **`null`** values, e.g., when the user want to clear attribute value by **`modify`** API method, he should set the attribute value to empty **`struct`** and the value will be cleared on the server.

2.1 Failure Related

Failures are propagated through XML-RPC by **`faultCode`** and **`faultString`** values. List of common faults:

- **`faultCode = 0`**
  Unknown fault which is described by **`faultString`**.
- **`faultCode = 1`**
  The `class` is not defined, the **`faultString`** specifies which `class`.
- **`faultCode = 2`**
  The attribute is not defined, the **`faultString`** specifies which attribute in which class.
- **`faultCode = 3`**
  The attribute type is wrong, the **`faultString`** specifies which attribute in which class and also it specifies the presented and required type.
- **`faultCode = 4`**
  The value of an enum attribute is wrong., the **`faultString`** specifies which value.

These are only common faults that are independent on specific API section. Other business logic faults can be generated and are described in appropriate API section.

2.2 Security and Identity Related

- **`class UserIdentity`**

Each user that accesses shongo or participates in shongo managed videoconference should be identified by **UserIdentity** definition.

Attributes:

- **String id** (**Required**)
  Equals to eduID identity. In future there can be unique identifier that associates multiple eduID for the same person.

- **class SecurityToken**

  Contains identity and credentials of a user performing the requested operation.

  Attributes:

  - **UserIdentity user** (**Required**)
    User identity.
  - **TODO:** Authorization data

2.3 Time Related

- **atomic_type Period = String**

  Represented as ISO8601 period (e.g., `P3Y6M4DT12H30M5S` which is *3 years, 6 months, 4 days, 12 hours, 30 minutes, and 5 seconds* or `P4W` which is *4 weeks*). The first character "P" means period and it comes from the ISO8601 standard. Components can be omitted (e.g., `P3YT12H` which is *3 years and 12 hours*). The zero duration is represented by `PT0S` value (which is *0 seconds*).

- **class DateTime**

  It serves only as base class for **AbsoluteDateTime**, **RelativeDateTime** and **PeriodicDateTime**.

- **class AbsoluteDateTime extends DateTime**

  Represented as ISO8601 date/time in UTC (e.g., 20120130T10:09:55). More efficient way of implementation may be used internally, of course. All time data are handled as in UTC while clients are responsible for suitable converting between timezones.

  Attributes:

  - **String dateTime** (**Required**)

  Example: We want to create a new reservation for resources at the precise date. We can specify it by **AbsoluteDateTime**:

  ```
  absoluteDateTime.dateTime = 20121231
  ```

- **class RelativeDateTime extends DateTime**

  Relative date is calculated as current date and time increased by **duration**.

  Attributes:

  - **Period duration** (**Optional**, default: `PT0S`)
    See **Period** for format specification.

  Example: We want to define the maximum amount of time since the request moment (e.g., the user can specify dates which are at most 4 months ahead). We can specify it by **RelativeDateTime** as follows:

```
relativeDateTime.duration = P4M
```

- **class PeriodicDateTime extends DateTime**

  It can be used for events that takes place repeatedly, but also for events that take place only once.

  Attributes:

  - **AbsoluteDateTime start** (**Required**)
    Defines the first occurrence of an event.
  - **Period period** (**Optional**)
    Defines the period in which the repeated events take place. See **Period** for format specification.
  - **AbsoluteDateTime end** (**Optional**)
    Ending date/time for events to not occur forever.
  - **Rule[] rules** (**Optional**)
    List of rules, which can define an extra events out of the periodicity or cancel specified periodical events. **Rule** can be one of the following types:
    - **Enable**/**Disable** event(s) in the specified **dateTime** or interval by **start** and **end**.
    - **Extra** event in the specified **dateTime**

    Rules contains implicit definition of **Enable** rule for whole **PeridiocDateTime** interval. Conflicts are solved by *last-match* policy.

  Example: Only one lecture on 20.3.2012.

  ```
  periodicDateTime.start = 20110908T12:00:00
  ```

  Example: A lecture on every Thursday at 12:00 with extra lecture on 20.3.2012 and Christmas holidays.

  ```
  periodicDate.start = 20110908T12:00:00
  periodicDate.period = P1W
  periodicDate.end = 20120631
  periodicDate.rules = {
      { type = Disable, from = 20111219, to = 20120101 },
      { type = Extra, dateTime = 20120320 }
  }
  ```

- **class DateTimeSlot**

  Date/time slot is a pair of **DateTime** and **Period**.

  Attributes:

  - **DateTime start** (**Required**)
    Defines the start of date/time slot.
  - **Period duration** (**Optional**)
    Defines the duration of date/time slot.

  For reservation purposes, the array **DateTimeSlot[]** should be used to provide the ability to reserve multiple date/times with different periods (e.g., on every Monday from 14:00 to 15:00 and every Thursday from 16:00 to 18:00).

  If date/time slot contains **PeriodicDateTime**, all periodic events can be listed by evaluating date/time slot to **AbsoluteDateTimeSlot[]**.

- **class** `AbsoluteDateTimeSlot` **extends** `DateTimeSlot`

  Extension of **`DateTimeSlot`** that allows only **`AbsoluteDateTime`** as an instance of date/time in the slot.

2.4 Reservations and Resources

- **enum** `ResourceType`

  The resource is one of the types as defined in UC-2.

  Enumeration values:

  - **`ManagedDevice`**
  - **`UnmanagedDevice`**
  - **`VirtualRoom`**
  - **`License`**
  - **`Identifier`**
  - **`PhysicalRoom`**
  - **`Other`**

- **enum** `Technology`

  Enumeration values:

  - **`H323`**
  - **`SIP`**
  - **`AdobeConnect`**
  - **`Skype`**
  - **`BigBlueButton`**
  - **`OpenMeeting`**
  - **`WebEx`**

- **class** `Resolution`

  Attributes:

  - **`Integer width`** (**Required**)
  - **`Integer height`** (**Required**)

  Enumeration values:

  - **`QCIF`** (176x144)
  - **`CIF`** (352x288)
  - **`4CIF`** (704x576)
  - **`720p`** (1280x720)
  - **`1080p`** (1920x1080)
  - **`2K`** (2048x1080)
  - **`4K`** (4096x2160)

- **enum** `AudioCodec`

  Enumeration values:

  - **`MP3`**
  - **`AC3`**

- **enum** `VideoCodec`

  Enumeration values:

  - `H261`
  - `H263`
  - `H264`

- **class** `Format`

  Attributes:

  - `Technology technology` (**Required**)
  - `AudioCodec audio` (**Required**)
  - `VideoCodec video` (**Required**)
  - `Resolution resolution` (**Required**)

  Can be represented by string:

  ```
  <technology>:<audio>:<video>:<resolution>
  ```

- **enum** `CapabilityType`

  Enumeration values:

  - `Input`
  - `Output`
  - `Multipoint`
    TODO: Does it make sense to have Multipoint capability type?
  - `Translation`
  - `Recording`
  - `Streaming`

- **class** `Capability`

  Attributes:

  - `CapabilityType type` (**Required**)
    Type of capability, see **`CapabilityType`**.
  - `Rule[] rules` (**Required**)
    List of rules that defines which formats is capability able to process. Rule can be one of the following types:
    * **`Enable`** means that the device is able to process specified format.
    * **`Disable`** means that the device is not able to process specified format.

    Conflicts in rules are solved by *last-match* policy. For **`Input`**, **`Output`**, **`Recording`** and **`Streaming`** capability type one **`format`** must be specified. For **`Translation`** capability type format **`from`** and **`to`** must be specified.

  Example:

  ```
  capability.type = Translation
  capability.rules = {
      { type = Enable, from = H323,SIP:*:H264:*, to = H323,SIP:*:H264:* },
      { type = Disable, from = *:AC3:*:*, to = *:AC3:*:* },
  }
  ```

- **`class Resource`**

  This class represents a complete resource definition. It contains child resources, possible translations and all other resource attributes. This class is used for creating and modifying resources.

  Attributes:

  - **`String id`** (**Required**)
    Resource unique identifier as defined in UC-4.
  - **`String parentId`** (**Required**)
    A parent resource identifier in which is the resource located (e.g., identifier of a physical room).
  - **`String name`** (**Required**)
    Short name which describes the resource.
  - **`ResourceType type`** (**Required**)
    Type of the resource, see **`ResourceType`**.
  - **`Capability[] capabilities`** (**Optional**)
    Capabilities of the resource.
  - **`String description`** (**Optional**)
    Long description depicting the resource.
  - **`boolean schedulable`** (**Optional**, default: `false`)
    Specifies whether the resource can be allocated to a reservation by a scheduler. When creating a new resource, it is useful to set **`schedulable`** to `false` and restrict the time when the resource can by used for public scheduling (e.g., setup permanent reservations) and then modify the **`schedulable`** to `true`.
  - **`DateTime maxFuture`** (**Optional**)
    The maximum future time for reservations as defined in UC-12.
  - **`String[] childResources`** (**Optional**)
    List of child resources identifiers (e.g., the resource is physical room and **`childResources`** contains all videoconferencing devices in the room).

- **`class ResourceSummary`**

  This class represents a summary of a resource. The summary of resource is lightweight and does not contain all resource attributes. It contains some useful additional "calculated" attributes. It is suitable when listing a lot of resources from controller database where the detail information about resource is not appropriate.

  Attributes:

  - **`String id`** (**ReadOnly**)
  - **`String parentId`** (**ReadOnly**)
  - **`String name`** (**ReadOnly**)
  - **`ResourceType type`** (**ReadOnly**)
  - **`String technology`** (**ReadOnly**)
    Enumeration of supported technologies
  - **`boolean schedulable`** (**ReadOnly**)
  - **`DateTime maxFuture`** (**ReadOnly**)

- **`enum ReservationType`**

  Enumeration values:

  - **`Periodic`**

One time or periodic reservation as defined in UC-8 and UC-9 (one time reservation is a special case of periodic reservation).
  - **`Permanent`**
    Permanent reservation as defined in UC-10.

- **`class Reservation`**

  This class represents a complete reservation definition. It contains specification of requested resources, requested date/time slots, child reservations and all other reservation attributes. This class is used for creating and modifying reservations.

  Attributes:

  - **`String id`** (**ReadOnly**)
    Reservation unique identifier as defined in UC-7.
  - **`ReservationType type`** (**ReadOnly**)
    Type of reservation, see **`ReservationType`**.
  - **`boolean interDomain`** (**Optional**, default: `false`)
    Specify whether the scheduler should try allocate resources from other domains.
  - **`String description`** (**Optional**)
    Long reservation description.
  - **`Resource[] resources`** (**Required**)
    List of requested resources by this reservation. Each **`Resource`** definition has filled the resource identifier (FQESpec) or other attributes that partially specifies the resource (PQESpec).
  - **`DateTimeSlot[] slots`** (**Required**)
    Requested date/time slots for the reservation.
  - **`String[] childReservations`** (**Optional**)
    List of child reservations identifiers. All allocated resources from child reservations become part of the parent reservation.
  - **`UserIdentity[] users`** (**Optional**)
    List of users that will be permitted to participate in the videoconference. The list can contain several empty **`User`** definitions to allow guests.

- **`class ReservationAllocation`**

  This class represents a successfully allocated date/time slots and its' allocated resources for a reservation. It contains only read only data that are obtained from a scheduler.

  Attributes:

  - **`String id`** (**ReadOnly**)
    Reservation unique identifier as defined in UC-7.
  - **`Map<AbsoluteDateTimeSlot, String[]> slotResources`** (**ReadOnly**)
    Map of allocated resources where key is allocated date/time slot and value is list of allocated resources identifiers for the date/time slot (for different date/time slots there can be different allocated resources).

- **`class ReservationSummary`**

  This class represents a summary of a reservation. The summary of reservation is lightweight and does not contain all reservation attributes. It contains some useful additional "calculated" attributes. It is suitable when listing a lot of reservations from controller database where the detail information about reservation is not appropriate.

Attributes:

- **`String id`** (**ReadOnly**)
  Reservation unique identifier as defined in UC-7.
- **`ReservationType type`** (**ReadOnly**)
- **`String description`** (**ReadOnly**)
- **`AbsoluteDateTime dateTime`** (**ReadOnly**)
  Specifies the first future date/time when the reservation takes place.

- **`class RoomUser`**

  Represents an active user in a virtual room on a server.

  Attributes:

  - **`String userId`** (**ReadOnly**)
    User identification in room (technology specific).
  - **`String roomId`** (**ReadOnly**)
    Room unique identifier. `TODO: The identifier should contain a part denoting the user and a part denoting his/her session - as requested by use case 24. resolved on the UserIdentity level`
  - **`UserIdentity userIdentity`** (**ReadOnly**)
    User identity which in some cases may be `null` (e.g., when the user is calling from cell phone).
  - **`AbsoluteDateTime joinTime`** (**ReadOnly**)
    Date and time when the user joined the room.
  - **`boolean muted`** (**ReadOnly**)
    Is the user muted?
  - **`int microphoneLevel`** (**ReadOnly**)
    Microphone level.
  - **`int playbackLevel`** (**ReadOnly**)
    Playback level (speakers volume)

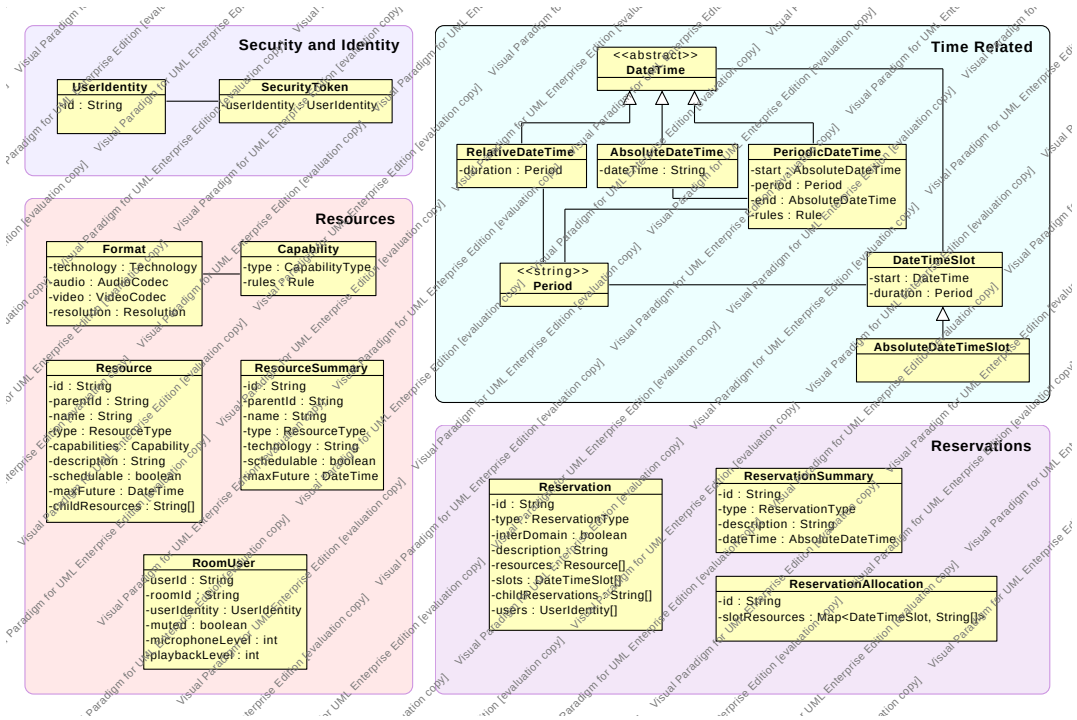

Figure 2.1: Class Diagram of Common Data Types

Chapter 3

User Interface API Specification

3.1 Data types

- **`enum ControllerType`**

  Types of controllers in the domain.

  Enumeration values:

  - **`Primary`**
    The primary controller.
  - **`Backup`**
    A backup controller. Behaves just like the primary controller, even some agents may be connected to a backup controller instead of the primary one.

- **`class ControllerInfo`**

  Represents information about the domain controller.

  Attributes:

  - **`String name`** (**Required**)
    A unique name within the domain.
  - **`String description`** (**Optional**)
  - **`ControllerType type`** (**Required**)

- **`class Agent`**

  Represents an agent managing a resource.

  Attributes:

  - **`String identifier`** (**Required**)
    A unique agent identifier within the domain.
  - **`String address`** (**Required**)
    An address where the agent is reachable.
  - **`int port`** (**Required**)
    Port on which the agent is reachable.
  - **`ResourceSummary resource`** (**Optional**)
    Resource managed by the agent.

- **class DomainInfo**

  Information about a domain and its controller.

  Attributes:

  - **String name** (**Required**)
    A unique domain name.
  - **String organization** (**Optional**)
    Organization owning the domain.
  - **ControllerInfo controller** (**Required**)
    A controller of the domain.
  - **boolean connected** (**Required**)
    Whether a connection to the domain controller is established.

3.2 Common

- **ControllerInfo getControllerInfo()**

  Get information about the domain controller. See **ControllerInfo** class.

- **Agent[] listAgents(Map filter)**

  Lists all agents within the platform (i.e., managed by the controller) managing a resource satisfying a given filter.

- **ControllerInfo[] listControllers()**

  Lists the primary and all backup controllers for the domain.

- **DomainInfo[] listDomains()**

  Lists all known domains.

3.3 Resources

- **String createResource(SecurityToken token, String domain, Map attributes)**

  Create a new resource that will be managed by Shongo. The new resource identifier is returned as a result. The user with given **token** will be the resource owner. Map of **attributes** should contain only attributes specified in **Resource** class and all attributes marked as **Required** must be present.

- **modifyResource(SecurityToken token, String resourceId, Map attributes)**

  Modify the resource with specified **resourceId**. That operation is permited only when the user with given **token** is the resource owner. Map of **attributes** should contain only attributes specified in **Resource** class.

- **deleteResource(SecurityToken token, String resourceId)**

  Delete the resource with specified **resourceId** from Shongo management. That operation is permited only when the user with given **token** is the resource owner and only when the resource is not used in any future reservation.

- **`Resource getResource(SecurityToken token, String resourceId)`**

  Get the complete resource object for specified **`resourceId`** that a user with given **`token`** is entitled to see. See **`Resource`** for details.

- **`ResourceSummary[] listResources(SecurityToken token, Map filter)`**

  List of resources managed by Shongo, that a user with given **`token`** is entitled to see and that meet the resource **`filter`** which contains only attributes defined in **`ResourceSummary`** class. See **`ResourceSummary`** for details.

- **`boolean isResourceActive(SecurityToken token, String resourceId, AbsoluteDateTime dateTime)`**

  Checks whether the resource with specified **`resourceId`** is in given **`dateTime`** used by any reservation.

- **`AbsoluteDateTimeSlot[] findResourceAvailableSlots(SecurityToken token, String resourceId)`**

  Lookup date/time slots when the resource with specified **`resourceId`** is not allocated to any reservation.

3.3.1 Failures

- **`faultCode = 100`**

TODO: Complete this

3.4 Reservations

- **`String createReservation(SecurityToken token, ReservationType type, Map attributes)`**

  Create a new reservation. The new reservation identifier is returned as a result. Map of **`attributes`** should contain only attributes specified in **`Reservation`** class and all attributes marked as **Required** must be present.

- **`modifyReservation(SecurityToken token, String reservationId, Map attributes)`**

  Modify the reservation with specified **`reservationId`**. Map of **`attributes`** should contain only attributes specified in **`Reservation`** class.

- **`deleteReservation(SecurityToken token, String reservationId)`**

  Release the reservation with specified **`reservationId`**. The child reservations remain untouched.

- **`Reservation getReservation(SecurityToken token, String reservationId)`**

  Get the reservation object for specified **`reservationId`** that a user with given **`token`** is entitled to see. The returned object contains requested time slots, requested resources, child reservations and all other attributes that can be modified. It does not contain the read only scheduler allocation information which can be obtained by **`getReservationAllocation`**.

- **`ReservationAllocation getReservationAllocation(SecurityToken token, String reservationId)`**

List all the date/time slots that were allocated by a scheduler for the reservation and for all child reservation (recursive). Each date/time slot contains list of identifiers for resources that are allocated for the date/time slot.

- **`ResourceSummary[] listReservationSlotResources(SecurityToken token, String reservationId, AbsoluteDateTimeSlot slot, Map filter)`**

  Get a list of allocated resources for the given date/time slot in a reservation with specified **`reservationId`** (one reservation can have multiple date/time slots in which the reservation takes place and the list of allocated resources may vary). The list of resources is filtered by specified **`filter`** map that should contain only attributes specified in **`ResourceSummary`**.

- **`ReservationSummary[] listReservations(SecurityToken token, Map filter)`**

  List all the reservations that a user with given **`token`** is entitled to see and that meet the reservation **`filter`** which contains only attributes defined in **`ReservationSummary`** class. Only the lightweight definitions of reservations is returned, see **`ReservationSummary`** for details.

- **`AbsoluteDateTimeSlot[] findReservationAvailableSlots(SecurityToken token, Period duration, Resource[] resources, boolean interDomain)`**

  Lookup available date/time slots for specified reservation **`duration`** and **`resources`**. Argument **`interDomain`** specifies whether inter-domain lookup should be performed.

3.4.1 Failures

- **`faultCode = 200`**

TODO: Complete this

3.5 Room Operations

- **`RoomUser[] listRoomUsers(SecurityToken token, String roomId)`**

  Get the list of users that currently participate in the room with specified **`roomId`**.

- **`RoomUser getRoomUser(SecurityToken token, String roomId, String userId)`**

  Gets a concrete room user.

- **`modifyRoomUser(SecurityToken token, String roomId, String userId, Map attributes)`**

  Modifies the user with specified **`userId`** in the room with given **`roomId`** (suitable for setting microphone/playback level, muting/unmuting...).

- **`disconnectRoomUser(SecurityToken token, String roomId, String userId)`**

  Disconnect the user with specified **`userId`** from the room with given **`roomId`**.

3.5.1 Failures

- **`faultCode = 300`**

TODO: Complete this

Chapter 4

Connector API Specification

4.1 Communication Protocol

TODO: Merge with XML-RPC description. Afterwards, just state, what is used for which communication acts (XML-RPC for user interface, Jade ontologies for Jade messaging). Use common failure codes. Groups of failure codes: Shongo-connections, connections between a connector and its device, failures reported by the devices themselves

Communication among controllers and connectors is implemented using JADE [3]. All messages are encoded using the FIPA SL content language [4]. An ontology, called **ShongoOntology**, is used by communicating agents to give the same meaning to the symbols used in messages. This section describes the way commands defined by this API are composed to messages and interpreted by Shongo agents.

The ontology used by all agents consists of concepts, predicates, and agent actions.

An agent action, tagged by **`jade.content.AgentAction`** interface, expresses a request what should the receiving agent do. Each of the commands specified in this API document is defined by a class implementing **`AgentAction`**, declaring all the command arguments as attributes accessed by public getters and setters.

A predicate, tagged by **`jade.content.Predicate`** interface, expresses a claim about a fact. In this API, we use just two predicates defined in the JADE framework, for the purpose of expressing result of a command. We use no custom predicates.

A concept, tagged by **`jade.content.Concept`** interface, is any entity which may be a part of an agent action or a predicate. All object types of arguments or return values must be specified as concepts for the agent content manager to be able to properly encode them in messages. In particular, any such a class must implement the **`jade.content.Concept`** interface and reside within the **`cz.cesnet.shongo.ontology`** package for the **`ShongoOntology`** class to be able to find it and comprise it in the ontology used for encoding messages.

For example, the **`setMicrophoneLevel(int level)`** command, defined in section 4.5, might be specified by the following class:

```
package cz.cesnet.shongo.ontology;

public class SetMicrophoneLevel implements AgentAction {
    private int level = 0;
```

```
    public int getLevel() {
        return level;
    }
    public void setLevel(int level) {
        this.level = level;
    }
}
```

The **setMicrophoneLevel** call implementation instantiates a new **SetMicrophoneLevel** object, sets up the **level** attribute, and passes the object to a controller agent content manager to send it to an endpoint as a **request** communicative act [5]. The corresponding endpoint agent creates the **SetMicrophoneLevel** object received from the controller agent and implements the requested functionality according to it. The message sent during such a call might be similar to the following:

```
(REQUEST
 :receiver  (set ( agent-identifier :name dev@127.0.0.1:1099/JADE ) )
 :content  "((action (agent-identifier :name
     Controller-Main-Container@127.0.0.1:1099/JADE :addresses (sequence
     http://localhost:7778/acc)) (SET-MICROPHONE-LEVEL :level 46)))"
 :language  fipa-sl  :ontology  shongo-ontology )
```

The agent receiving a command should always send a reply as an **inform** [5] message. In case of commands without any return value, a **Done** predicate from the package **jade.content.onto.basic** should be sent as a reply, denoting a successful command execution. When a return value is expected, a **Result** predicate, defined in [4], is sent, filled with the value to be returned. The same requirements apply to the class of the object to be returned as for command object arguments – the class must reside within the **cz.cesnet.shongo.ontology** interface and be tagged by the **Concept** interface.

An example of a complex command is shown in appendix B.

4.2 Data Types

- **class ConnectorInfo**

  Information about connector.

  Attributes:

  - **String name** (**ReadOnly**)
    the connector name
  - **Resource device** (**ReadOnly**)
    the device managed by this connector (must be a resource of type ManagedDevice – see chapter 2.4)
  - **ConnectionState connectionState** (**ReadOnly**)
    connection state to the device
  - **DeviceState deviceState** (**ReadOnly**)
    state of the device, maintained by the connector for performance reasons

- **enum ConnectionState**

  State of connection between a connector and a device it manages.

Enumeration values:

- `Connected`
- `Disconnected`

- **`class DeviceState`**

  State description of a device. TODO

- **`class DeviceLoadInfo`**

  Current device load information. A negative value in any attribute means the value could not be determined.

  Attributes:

  - `Float cpuLoad` (ReadOnly)
  - `Long memoryOccupied` (ReadOnly)
  - `Long memoryAvailable` (ReadOnly)
  - `Long diskSpaceOccupied` (ReadOnly)
  - `Long diskSpaceAvailable` (ReadOnly)

- **`class Room`**

  Represents a virtual room on a multipoint server device.

  Attributes:

  - `RoomUser[] users` (Required)
    List of allowed users. There should be option to also allow guests.
  - `int licenseCount` (Required)
    Number of licenses that multipoint server can utilize for this room.
  - `String[] configuration` (Optional)
    Platform specific configuration.

  TODO: Room settings should be auto-modified in time be uploaded calendar

- **`class UsageStats`**

  Usage stats of a given multipoint device.

  Attributes:

  - `Byte[] callLog` (ReadOnly)
    Call log in CDR. Should contain at least start time and duration of each call.

- **`class RoomInfo`**

  A brief info about a virtual room at a server.

  Attributes:

  - `String name` (Required)
    Name of the room.
  - `String owner` (ReadOnly)
    Identification of the room owner.
  - `AbsoluteDateTime creation` (ReadOnly)
    Date and time when the room was created.
  - `Reservation reservation` (ReadOnly)
    Reservation for which this room was created (to satisfy use-case 23).
  - `Technology type` (ReadOnly)
    Type of the room.

- `class MediaData`

  Custom media data, typically used for uploading or downloading some content (images, documents, etc.).

  Attributes:

  - `ContentType contentType` (**Required**)
    Type of the data.
  - `byte[] data` (**Required**)
    The content. To be interpreted according to the content type.

- `class ContentType`

  Description of a media type. Any MIME Media Type listed by IANA [?], e.g. `image/jpeg`.

  Attributes:

  - `String type` (**Required**)
    Textual name of the type (e.g., **`image`** or **`text`**).
  - `String subtype` (**Required**)
    Textual name of the subtype (e.g., **`jpeg`** or **`html`**).

4.3 Common API

- `ConnectorInfo getConnectorInfo()`

  Get information about connector.

- `muteRoomUser(SecurityToken token, String RoomUserId)`

  Mutes a user in a room.

- `unmuteRoomUser(SecurityToken token, String RoomUserId)`

  Unmutes a user in a room.

- `setMicrophoneLevel(SecurityToken token, String RoomUserId, int level)`

  Sets microphone audio level of a user in a room to a given value. Note that the implementation differs between multipoint and endpoint types of devices. On an endpoint, the playback level is set using the device amplifier, while calling this on a multipoint device results in software adaptation of the output sound data (which may result in a distorted sound).

- `setPlaybackLevel(SecurityToken token, String RoomUserId, int level)`

  Sets playback audio level of a user in a room to a given value. Note that the implementation differs between multipoint and endpoint types of devices. On an endpoint, the playback level is set using the device amplifier, while calling this on a multipoint device results in software adaptation of the output sound data (which may result in a distorted sound).

- `enableUserVideo(SecurityToken token, String RoomUserId)`

  Enables video from a user in a room.

- **disableUserVideo(SecurityToken token, String RoomUserId)**

  Disables video from a user in a room.

  TODO: How to upload/modify calendars for Endpoint/Multipoint device types? connectors not involved in calendars regarding reservations; this TODO item mentions rather a plan what to do to prevent the need for immediate commands just on time of an event

4.4 Multipoint Device

4.4.1 Room Management

- **RoomInfo getRoomInfo(SecurityToken token, String roomId)**

  Gets info about an existing room.

- **String createRoom(SecurityToken token, Room room)**

  Create a new virtual room on a multipoint device that is managed by this connector. The settings are specified in **room** definition. TODO: the value returned is the roomId?

- **deleteRoom(SecurityToken token, String roomId)**

  Delete an existing virtual room on a multipoint device that is managed by this connector.

  TODO: room layout setup - specified by use case 26 and 35; see the Tandberg and Polycom specification (API) and design a class for both; layout config: windows positions, voice-switched; a default layout for room, and user-defined - ordered by the user for the multipoint to send to them; in the worst case, define our reasonable set of fixed layouts and offer just them

- **String exportRoomSettings(SecurityToken token, String RoomId)**

  Gets current settings of a room exported to a textual format. TODO: What format? XML? Should we specify the corresponding XML Schema somewhere? room name, technology (H.323/SIP/Connect...) settings, a need for structure versioning; specify in RelaxNG

- **importRoomSettings(SecurityToken token, String RoomId, String settings)**

  Sets up a room according to a given configuration previously exported by the **exportRoomSettings** method. TODO: Describe the settings argument according to resolution of exportRoomSettings TODO item

4.4.2 User Management

- **RoomUser[] listRoomUsers(SecurityToken token, String roomId)**
- **RoomUser getRoomUser(SecurityToken token, String roomId, String roomUserId)**

  Get user information and settings in a room.

- **modifyRoomUser(SecurityToken token, String roomId, String roomUserId, Map attributes)**

  Modify user settings in the room (suitable for setting microphone/playback level, muting/unmuting...)

- **`disconnectRoomUser(SecurityToken token, String roomId, String roomUserId)`**

  Disconnect user from the room.

- **`enableContentProvider(SecurityToken token, String roomUserId)`**

  Enables a given room user as a content provider in the room. This is typically enabled by default.

- **`disableContentProvider(SecurityToken token, String roomUserId)`**

  Disables a given room user as a content provider in the room. Typically, all users are allowed to fight for being the content provider. Using this method, a user is not allowed to do this.

4.4.3 Room Content Management

- **`Byte[] getRoomContent(SecurityToken token, String roomId)`**

  Gets all room content (e.g., documents, notes, polls, etc.) as a single **`tgz`** archive. **TODO:** Use a tgz archive? Is the room content in form of plain files just accessible to room members, or are there some special kinds of content?

- **`addRoomContent(SecurityToken token, String roomId, String name, Byte[] data)`**

  Adds a data file to room content under a given name. **TODO:** Are there some special kinds of content? **TODO:** Use compression?

- **`removeRoomContentFile(SecurityToken token, String roomId, String name)`**

  Removes a file of a given name from room content. **TODO:** Is filename enough? yes, typically just a plain storage without directory structure

- **`clearRoomContent(SecurityToken token, String roomId)`**

  Clears all room content.

4.4.4 Monitoring

- **`DeviceLoadInfo getDeviceLoadInfo()`**

  Gets info about current load of the device.

- **`UsageStats getUsageStats()`**

  Gets the multipoint usage stats.

- **`RoomInfo[] getRoomList()`**

  Gets a list of all rooms at a given server.

- **`MediaData getReceivedVideoSnapshot(SecurityToken token, String RoomUserId)`**

  Gets a snapshot of the video stream received by a user in a room. The return value is a byte sequence – the resulting image. **TODO:** (for getSentVideoSnapshot, too) add ability to prefer some image format, or is it determined by the device?

- **`MediaData getSentVideoSnapshot(SecurityToken token, String RoomUserId)`**

  Gets a snapshot of the video stream that a user is sending in a room. The return value is a byte sequence – the resulting image.

4.4.5 Recording

- **`int startRecording(SecurityToken token, String roomId, ContentType format)`**

  Immediately starts recording in a given room to a given format. Returns an identifier for further reference, unique among other recordings on the device. Does not have any effect and returns 0 if the room is already being recorded. **TODO:** (for scheduleRecording, too) what does "format: PIP, side by side" mean? (answered at the previous meeting) Does it specify the layout using which the room will be recorded?

- **`int scheduleRecording(SecurityToken token, String roomId, ContentType format, DateTimeSlot dateTimeSlot)`**

  Schedules recording in a given room to a given format. On the given time, the device should connect to the room and start recording for the given period. Returns an identifier for further reference, unique among other recordings on the device. Does not have any effect and returns 0 if the room is already being recorded.

- **`stopRecording(SecurityToken token, int recordingId)`**

  Stops recording. The **`recordingId`** parameter, specifying what to stop, is an identifier previously returned by **`startRecording`** or **`scheduleRecording`**.

- **`String getRecordingDownloadURL(SecurityToken token, int recordingId)`**

  Returns a URL from where it is possible to download a recording. The **`recordingId`** parameter is an identifier previously returned by **`startRecording`** or **`scheduleRecording`**.

- **`notifyParticipants(SecurityToken token, int recordingId)`**

  Sends an e-mail to all non-anonymous participants present in the room recorded. If the room is still being recorded, the operation is deferred to the moment when the recording stops. **TODO:** "...participants present in the room..." - present when? Any participant present in any time of the reservation? Or any participant present in any moment of the recording? Someone has to manage such an information. Or just the registered users listed within the reservation currently being realized in the room?

- **`uploadRecording(SecurityToken token, int recordingId, String targetURL)`**

  Starts uploading a recording to a destination given by a URL (note it may include credentials needed to upload it there). The **`recordingId`** parameter is an identifier previously returned by **`startRecording`** or **`scheduleRecording`**. Returns immediately when the uploading starts. Management of recordings on the destination server is out of scope of this device. **TODO:** correct, that "Management of recordings on the destination server is out of scope of this device."?
  TODO: Resolve the case when an upload fails. Consider the situation when the controller calls uploadRecording and deleteRecording - the recording could then be unrecoverably deleted if the uploading failed. Maintain a state of the recording (e.g., "being uploaded") and qualify the deletion by such a state? Or return from uploadRecording only when the upload finishes (successively or not)? Or let the device send a message about the uploading result - then, a reasonable controller would call deleteRecording only upon a successful uploading.

- **`deleteRecording(SecurityToken token, int recordingId)`**

Deletes a given recording. The **recordingId** parameter is an identifier previously returned by **startRecording** or **scheduleRecording**. If the recording is being worked with somehow (still being recorded, being uploaded, etc.), the operation is deferred to the moment when current operations are completed.

4.5 Endpoint Device

- **dial(SecurityToken token, String server)**

  Dials a server. TODO: dial should be common to endpoints and multipoints, since even MCU may dial an endpoint - correct? yes (even it is available only on some technologies, e.g. an H.323/SIP multipoint is able to dial the client, Connect is out of ability -> leave here and copy to 4.7

- **resetDevice(SecurityToken token)**

  Resets the device.

4.6 Technology Specific API

TODO: Cover use cases 38 and 39.
TODO: How to structure this section? List the supported commands for each technology separately, or list them on a single place, stating the technologies supporting a functionality for each command?

- **dial(SecurityToken token, String deviceAddress)**

  Dials a device, multipoint or endpoint. Dialing a client is available only on **H.323** and **SIP**.

Chapter 5

Inter-Controller API Specification

Appendix A

User Interface API Usage

A.1 Perl programming language

A.1.1 Connect to Controller

```
#!/usr/bin/perl

require RPC::XML;
require RPC::XML::Client;

$client = RPC::XML::Client->new('http://localhost:8008');

$response = $client->send_request(...);

if ( ref($response) ) {
    use XML::Twig;
    $xml = XML::Twig->new(pretty_print => 'indented');
    $xml->parse($response->as_string());
    $xml->print();
} else {
    print($response . "\n");
}
```

A.1.2 Create reservation

```
$response = $client->send_request(
    'Reservations.createReservation',
    RPC::XML::struct->new(
        'class' => RPC::XML::string->new('SecurityToken'),
        ...
    ),
    RPC::XML::string->new('OneTime'),
    RPC::XML::struct->new(
        'date' => RPC::XML::struct->new(
            'class' => RPC::XML::string->new('Date'),
            'date' => RPC::XML::string->new('20120101')
        )
    )
);
```

Response

```
<struct>
  <member>
    <name>class</name>
    <value><string>Reservation</string></value>
  </member>
  <member>
    <name>id</name>
    <value>
      <string>e5a6ee96-8ac5-46dc-ac3b-5374076aee1b</string>
    </value>
  </member>
  <member>
    <name>type</name>
    <value><string>OneTime</string></value>
  </member>
  <member>
    <name>date</name>
    <value><struct>
        <member>
          <name>class</name>
          <value><string>Date</string></value>
        </member>
        <member>
          <name>date</name>
          <value><string>20120101</string></value>
        </member>
    </struct></value>
  </member>
</struct>
```

A.1.3 Modify reservation

```
$response = $client->send_request(
    'Reservations.modifyReservation',
    RPC::XML::struct->new(
        'class' => RPC::XML::string->new('SecurityToken'),
        ...
    ),
    RPC::XML::string->new('15082783-5b6f-4287-9015-3dbc0ab2f0d9'),
    RPC::XML::struct->new(
        'description' => RPC::XML::struct->new() # set description to null
    )
);
```

Response

```
<struct>
  <member>
    <name>id</name>
    <value><string>15082783-5b6f-4287-9015-3dbc0ab2f0d9</string></value>
  </member>
  <member>
    <name>class</name>
    <value><string>Reservation</string></value>
  </member>
  <member>
    <name>type</name>
    <value><string>OneTime</string></value>
  </member>
</struct>
```

A.1.4 List reservations

```
$response = $client->send_request(
    'Reservations.listReservations',
    RPC::XML::struct->new(
        'class' => RPC::XML::string->new('SecurityToken'),
        ...
    )
);
```

Response

```
<array><data>
  <value><struct>
    <member>
      <name>class</name>
      <value><string>Reservation</string></value>
    </member>
    <member>
      <name>id</name>
      <value><string>15082783-5b6f-4287-9015-3dbc0ab2f0d9</string></value>
    </member>
    <member>
      <name>type</name>
      <value><string>Periodic</string></value>
    </member>
  </struct></value>
</data></array>
```

A.1.5 Exception handling

Wrong class

```
$response = $client->send_request(
    'Reservations.listReservations',
    RPC::XML::struct->new(
        'class' => RPC::XML::string->new('SecurityTokenX'),
        ...
    )
);
```

Response

```
<fault>
  <value><struct>
    <member>
      <name>faultString</name>
      <value><string>Class 'SecurityTokenX' is not defined.</string></value>
    </member>
    <member>
      <name>faultCode</name>
      <value><i4>1</i4></value>
    </member>
  </struct></value>
</fault>
```

Wrong attribute name

```
$response = $client->send_request(
    'Reservations.listReservations',
    RPC::XML::struct->new(
        'class' => RPC::XML::string->new('SecurityToken'),
        ...
    ),
    RPC::XML::struct->new(
        'typeX' => RPC::XML::string->new('OneTime')
    )
);
```

Response

```
<fault>
  <value><struct>
    <member>
      <name>faultString</name>
      <value><string>Attribute 'typeX' in class 'Reservation' is not defined.</string></value>
    </member>
    <member>
      <name>faultCode</name>
      <value><i4>2</i4></value>
    </member>
  </struct></value>
</fault>
```

Wrong attribute value

```
$response = $client->send_request(
    'Reservations.listReservations',
    RPC::XML::struct->new(
        'class' => RPC::XML::string->new('SecurityToken'),
        ...
    ),
    RPC::XML::struct->new(
        'type' => RPC::XML::struct->new(
            'class' => RPC::XML::string->new('Date'),
            'date' => RPC::XML::string->new('20120101')
        )
    )
);
```

Response

```
<fault>
  <value><struct>
    <member>
      <name>faultString</name>
      <value><string>Attribute 'type' in class 'Reservation' has type
          'ReservationType' but 'Date' was presented.</string></value>
    </member>
    <member>
      <name>faultCode</name>
      <value><i4>3</i4></value>
    </member>
  </struct></value>
</fault>
```

Wrong enum

```
$response = $client->send_request(
    'Reservations.listReservations',
    RPC::XML::struct->new(
        'class' => RPC::XML::string->new('SecurityToken'),
        ...
    ),
    RPC::XML::struct->new(
        'type' => RPC::XML::string->new('OneTimeX')
    )
);
```

Response

```
<fault>
  <value><struct>
    <member>
      <name>faultString</name>
      <value><string>Enum value 'OneTimeX' is not defined in enum
          'ReservationType'.</string></value>
    </member>
    <member>
```

```
      <name>faultCode</name>
      <value><i4>4</i4></value>
    </member>
  </struct></value>
</fault>
```

Bussiness logic exception

```
$response = $client->send_request(
    'Reservations.createReservation',
    RPC::XML::struct->new(
        'class' => RPC::XML::string->new('SecurityToken'),
        ...
    ),
    RPC::XML::struct->new(
        'type' => RPC::XML::string->new('Periodic'),
        'date' => RPC::XML::struct->new(
            'class' => RPC::XML::string->new('Date'),
            'date' => RPC::XML::string->new('20120101')
        )
    )
);
```

Response

```
<fault>
  <value><struct>
    <member>
      <name>faultString</name>
      <value><string>Periodic date is required.</string></value>
    </member>
    <member>
      <name>faultCode</name>
      <value><i4>102</i4></value>
    </member>
  </struct></value>
</fault>
```

Appendix B

JADE Command Encoding Example

Consider the following command required by this API:

- **`RoomUser[] listRoomUsers(SecurityToken token, String roomId)`**

The following classes should be defined to represent the command and all objects used by it:

```
package cz.cesnet.shongo.ontology;

public class ListRoomUsers implements AgentAction {
    private SecurityToken token;
    private String roomId;

    public String getRoomId() {
        return roomId;
    }
    public void setRoomId(String roomId) {
        this.roomId = roomId;
    }
    public String getToken() {
        return token;
    }
    public void setToken(String token) {
        this.token = token;
    }
}

public class SecurityToken implements Concept {
    private UserIdentity user;

    public UserIdentity getUser() {
        return user;
    }
    public void setUser(UserIdentity user) {
        this.user = user;
    }
```

```
}

public class UserIdentity implements Concept {
    private String id;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
}

public class RoomUser implements Concept {
    private String userId;
    private String roomId;
    private UserIdentity userIdentity;
    private boolean muted;
    private int microphoneLevel;
    private int playbackLevel;

    // getters and setters ...
}
```

The command might be encoded in the following message:

```
(REQUEST
 :receiver  (set ( agent-identifier :name dev@127.0.0.1:1099/JADE ) )
 :content  "((action (agent-identifier :name
     Controller-Main-Container@127.0.0.1:1099/JADE :addresses
     (sequence http://localhost:7778/acc)) (ListRoomUsers)))"
 :language  fipa-sl  :ontology  shongo-ontology )
```

A successful reply would then be encoded as follows:

```
(INFORM
 :sender  ( agent-identifier :name dev@127.0.0.1:1099/JADE  :addresses (sequence
     http://localhost:7778/acc ))
 :receiver  (set ( agent-identifier :name Controller-Main-Container@127.0.0.1:1099/JADE
     :addresses (sequence http://localhost:7778/acc )) )
 :content  "((result (action (agent-identifier :name
     Controller-Main-Container@127.0.0.1:1099/JADE :addresses (sequence
     http://localhost:7778/acc)) (ListRoomUsers)) (sequence (RoomUser :microphoneLevel 45
     :muted false :playbackLevel 0 :roomId konf :userId Azurit :userIdentity
     (UserIdentity :id shongololo)) (RoomUser :microphoneLevel 57 :muted false
     :playbackLevel 0 :roomId konf :userId Shongololo :userIdentity (UserIdentity)))))"
 :reply-with  Controller-Main-Container@127.0.0.1:1099/JADE1336527079398  :language
     fipa-sl  :ontology  shongo-ontology )
```

Bibliography

[1] R. Moats, URN Syntax, RFC 2141 (May 1997).
URL http://www.ietf.org/rfc/rfc2141.txt

[2] P. Leach, M. Mealling, R. Salz, A Universally Unique IDentifier (UUID) URN Namespace, RFC 4122 (July 2005).
URL http://www.ietf.org/rfc/rfc4122.txt

[3] Java Agent Development Framework (May 2012).
URL http://jade.tilab.com

[4] Foundation for Intelligent Physical Agents, FIPA SL Content Language Specification, FIPA00008 (December 2002).
URL http://www.fipa.org/specs/fipa00008/

[5] Foundation for Intelligent Physical Agents, FIPA Communicative Act Library Specification, FIPA00037J (December 2002).
URL http://www.fipa.org/specs/fipa00037/