API for Shongo

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

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 URN standard [1]:

`urn:shongo:<type>:<domain>(.<subdomain>)*:(<section>.)*<name>`

The `<type>` component represents **`type`** of an entity (e.g., `resource`, or `reservation`). The `<domain>(.<subdomain>)*` 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 `(<section>.)*<name>` component represents entity identification inside the **`domain`**.

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:

`urn:shongo:resource:<domain>:<name>`

Each resource has own **`name`** and belongs to some main **`domain`** that runs controller platform. The **`domain`** name can be structured to any number of subdomains by "`.`" sign. The resource **`name`** can be also structured by "`.`" sign (e.g., to specify sections, or buildings). For unmanaged resources that don't belong to any domain there is a default `unmanaged` domain.

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:

`urn:shongo:reservation:<domain>:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx`

Each reservation is created in a **`domain`** that runs it's own controller platform. The **`xx...xx`** 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.).

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 including 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.

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 TimeSlot = Pair<DateTime, Period>**

  Time slot is a pair of **DateTime** and **Period**. For reservation purposes, the array **TimeSlot[]** should be used to provide the ability to reserve multiple date and 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 time slot contains **PeriodicDateTime**, all periodic events can be listed by evaluating time slot to **TimeSlot[]**, where only **AbsoluteDateTime**s are used.

2.4 Reservations and Resources

- **enum ResourceType**

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

Enumeration values:

- **ManagedEndPoint**
- **UnmanagedEndPoint**
- **MultipointServer**
- **GatewayServer**
  TODO: Perhaps remove GatewayServer (every server can act as gateway)
- **RecordingServer**
- **StreamingServer**
- **VirtualRoom**
- **License**
- **Identifier**
- **Other**
  A physical room for instance.

- **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)
    TODO: Complete this

- **enum** **CodecType**

  Enumeration values:

  - **H261**
  - **H263**
  - **H264**
    TODO: Complete this

- **class** **Codec**

  Resource technologies can provide different codecs.

Attributes:

- **`CodecType type`** (**Required**)
- **`Resolution resolution`** (**Required**)

- **`class Translation`**

  Represents translations that a **`GatewayServer`** resource can perform.

  Attributes:

  - **`Map<TechnologyType, Codec[]> input`** (**Required**)
    Input technologies with specified codecs that server is able to transform.
  - **`Map<TechnologyType, Codec[]> output`** (**Required**)
    Output technologies with specified codecs that server is able to produce.

  `TODO: Do better translation specification (add exceptions or replace by patterns)`
  `TODO: Specify also technology here and remove it from resource`
  `TODO: Rename it to something better`

- **`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`**.
  - **`Technology technology`** (**Optional**)
    Technology type of the resource, see **`Technology`**.
    `TODO: Perhaps remove technology or change it to array`
  - **`Translation translation`** (**Optional**)
    Translation type of the gateway resource, see **`Translation`**.
    `TODO: Change it according to Translation changes`
  - **`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[] resources`** (**Optional**)
    List of child resources identifiers (e.g., the resource is physical room and **`resources`** 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 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).
  - **`TimeSlot[] slots`** (**Required**)
    Requested time slots for the reservation.
  - **`String[] reservations`** (**Optional**)
    List of child reservations identifiers. All allocated resources from child reservations become part of the parent reservation.
  - **`User[] 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 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<TimeSlot, String[]> slotResources` (**ReadOnly**)
    Map of allocated resources where key is allocated time slot and value is list of allocated resources identifiers for the time slot (for different 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 id` (**ReadOnly**)
    User identification in room (technology specific).
  - `String roomId` (**ReadOnly**)
    Room unique identifier.
  - `UserIdentity user` (**ReadOnly**)
    User identity which in some cases may be `null` (e.g., when the user is calling from cell phone).
  - `boolean muted` (**ReadOnly**)
    Is the user muted?
  - `int microphoneLevel` (**ReadOnly**)
    Microphone level.
  - `int playbackLevel` (**ReadOnly**)
    Playback level (speakers volume)

Chapter 3

User Interface API Specification

3.1 Resources

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

  Create a new resource that will be managed by Shongo. 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.

- **`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.

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

  Checks whether the resource with specified **`resourceId`** is currently used by any reservation.

3.1.1 Failures

- **`faultCode = 100`**

TODO: Complete this

3.2 Reservations

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

  Create a new reservation. The new reservation identifier is included in returned object as **`id`** attribute. 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 complete reservation object for specified **`reservationId`** that a user with given **`token`** is entitled to see.

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

  List all the time slots with assigned resources that were allocated by a scheduler for the reservation and for all child reservation (recursive).

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

  Get the list of allocated resources for a reservation with specified **`reservationId`** and for the given time **`slot`** (one reservation can have multiple time slots in which the reservation takes place). 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.

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

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

3.2.1 Failures

- **`faultCode = 200`**

TODO: Complete this

3.3 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 a given room user (suitable for setting microphone/playback level, muting/unmuting...).
- **`disconnectRoomUser(SecurityToken token, String roomId, String userId)`**
  Disconnect the specified **`user`** from a given room.

3.3.1 Failures

- **`faultCode = 300`**

TODO: Complete this

Chapter 4

Connector API Specification

4.1 Data Types

4.2 Common API

4.3 Reservation API

4.4 User Management API

4.5 Monitoring API

4.6 Application Specific API

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>
```

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`