Shongo API

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

Contents

List of Terms

Shongo	Represents a distributed multimedia resource management system.
domain	Represents an organization which can run it's own controller and participate in Shongo.
controller	Represents an application that holds a database of resources, reservation requests and reservations for a single domain. Users can access and modify the database through a controller client. The controller runs a scheduler which allocates reservation requests to reservations.
controller client	Represents an application (e.g., command-line or web) which is able to connect to a controller and perform commands through the controller's API.
resource	An entity that can be requested for a reservation and allocated by a scheduler.
device	Represents a video/web conferencing hardware or software equipment (e.g., H.323 terminal, H.323 MCU, Adobe Connect server, gateway or streaming server).
device resource	A special type of resource that represents a device.
reservation request	A request that is made by an user to book resource(s) for a specific date/time slot(s). Reservation request can be *incomplete* or *complete*. Incomplete requests must be filled in by additional information to become complete. Scheduler processes only complete reservation requests.
reservation	Represents allocated resource(s) for a complete reservation request. Reservations are allocated by a scheduler.
user	Represents an authenticated person that can access a controller through a controller client.
scheduler	Scheduler is a component of a controller which processes complete reservation requests and allocates them to reservations.
connector	Represents an application that is composed of one or multiple connector agents.
connector agent	Represents a component that manages a single device resource and provides an API that allows the controller to access that equipment.

technology	Represents a single video/web conferencing technology (e.g., H.323, SIP, or Adobe Connect).
compartment	Represents a group of endpoints and/or persons which participates in a single video/web conference and the conference can spread out through multiple virtual rooms and even through multiple technologies (when a gateway device is used).
endpoint	Represents a device which can participate in a compartment (e.g., a H.323 terminal or SIP client).
virtual room	Represents a virtual room in a device (e.g., virtual room in H.323 MCU).
alias	Represents an identifier that can be assigned to a device and which can be used to connect to the device in a specific technology (e.g., E.164 number in H.323 or URI in SIP).
video conference call	Represents a connection or call which is initiated by a device to another device.
executor	Executor is a component of a controller which executes allocated compartments from reservations. It creates all virtual rooms on managed device resources and establishes all video conference calls between created virtual rooms and managed endpoint device resources.
preprocessor	Preprocessor is a component of a controller which allocates permanent reservation requests and creates single reservation requests from compound requests. Single reservation requests then can be allocated by the scheduler.

Chapter 1

Use Cases

1.1 Common

UC-1 (com:identification) Entity identification

Each entity in Shongo (e.g., resource, reservation request or reservation) is identified by an unique identifier. The identifier follows the URI standard [1]:

`shongo:<domain>:<id>`

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. The **`<id>`** component represents an identification of the entity in the specified 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 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 automatically assigned to the resource when it is being created.

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 when it is being created.

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

TODO: How does this use-case differ from use-case 26? Is this use-case a subset of use-case 26, which mentions also user-specific layout?

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

Controller Implementation

This chapter describes the implementation for database of resources and allocation of reservation requests to reservations in controller from the API point of view.

2.1 Resource database

Each controller contains persistent database of resources. New resources can be added to the controller's database and existing resources can be modified or deleted through the controller's API. A running controller holds it's database of resources in memory and when it is restarted it reloads the database from a persistent storage. Class diagram for resources that are persisted in the database is shown in fig. 2.1.

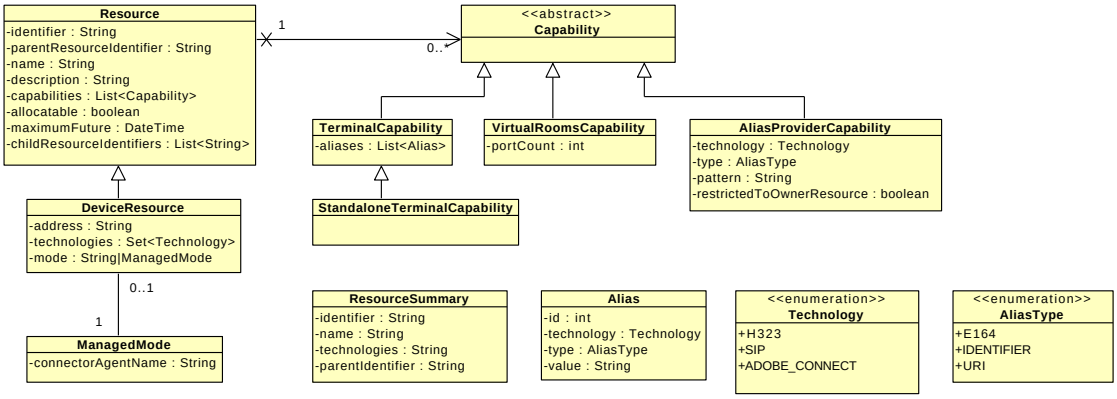

Figure 2.1: Class diagram for resource database in the controller API

A resource can be created by an instance of **`Resource`** or **`DeviceResource`** class. Each resource is identified by an unique **`identifier`** which is generated by the controller when the resource is created in the database. Each resource must have specified if it is **`allocatable`**. If a resource isn't **`allocatable`** a scheduler cannot allocate it to a reservation. A resource can also have specified **`maximumFuture`** which restrict the controller's scheduler to allocate the resource only to maximum date/time computed as current date/time plus specified **`maximumFuture`**. When a resource is

requested for a reservation which exceeds the maximum date/time, the scheduler will report an error.

Each resource can have specified zero, one or more **capabilities**. Capabality specifies that a resource provide some feature or service. Some capabilities can be used only for a **DeviceResource** (e.g., **TerminalCapability** or **VirtualRoomsCapability**). Complete description of capabilities can be found in chapter 4, but here is a short summary:

- **TerminalCapability** tells that a device resource can participate in a compartment (must be connected into a virtual room, e.g., Adobe Connect client).
- **StandaloneTerminalCapability** tells that a device resource can participate in a compartment. The compartment may consist only of two endpoints without a virtual room (e.g., H.323 terminal).
- **VirtualRoomsCapability** tells that a device resource can host multiple virtual rooms.
- **AliasProviderCapability** tells that a resource can allocate an alias which then can be assigned to a managed endpoint or a virtual room.

Here are some examples of resources which can be added to a resource database in a controller, e.g., for `cz.cesnet` domain:

- Codian MCU which can host multiple H.323 virtual rooms (the limit is 20 ports for all rooms) and which can assign aliases in range `9500872XX` to hosted virtual rooms.

```
mcu = DeviceResource {
  identifier: shongo:cz.cesnet:1,
  name: mcu,
  description: Codian MCU 4515,
  allocatable: true,
  maximumFuture: Period(P4M),
  technologies: [H323],
  capabilities: [
    VirtualRoomsCapability {
      portCount: 20
    },
    AliasProviderCapability {
      technology: H323,
      type: E164,
      pattern: 9500872[dd],
      restrictedToOwnerResource: true
    }
  ],
  mode: ManagedMode{
    connectorAgentName: codian
  }
}
```

- Endpoint Tandberg Codec C90 can participate in H.323 compartments and it has assigned a H.323 alias `950081038`.

```
endpoint = DeviceResource {
  identifier: shongo:cz.cesnet:2,
  name: endpoint,
  description: Tandberg Codec C90,
  allocatable: true,
  maximumFuture: Period(P4M),
  technologies: [H323],
  capabilities: [
    StandaloneTerminalCapability {
      aliases: [
        Alias {
          technology: H323,
          type: E164,
          value: 950081038,
        }
      ]
    }
  ],
  mode: ManagedMode{
    connectorAgentName: c90
  }
}
```

- Resource representing a physical lecture room.

```
room = Resource {
  identifier: shongo:cz.cesnet:3,
  name: room,
  description: Lecture room,
  allocatable: true,
}
```

- Another Tandberg endpoint which is located inside the **room** (`shongo:cz.cesnet:3`) and when the endpoint is allocated the room must be also allocated and thus when the room cannot be allocated, the endpoint fails to allocate.

```
anotherEndpoint = DeviceResource {
  identifier: shongo:cz.cesnet:4,
  parentIdentifier: shongo:cz.cesnet:3,
  ...
  technologies: [H323],
  capabilities: [
    StandaloneTerminalCapability {
      ...
    }
  ],
  ...
}
```

- Resource that can allocate an alias (H.323 phone number or SIP URI) which then can be assigned to any managed endpoint or virtual room in an allocated compartment.

```
aliasProvider = Resource {
  identifier: shongo:cz.cesnet:5,
  name: aliasProvider,
  description: Provider for H323 and SIP aliases,
  allocatable: true,
  maximumFuture: Period(P2Y)
  capabilities: [
    AliasProviderCapability {
      technology: H323,
      type: E164,
      pattern: 9500873[dd],
    }
    AliasProviderCapability {
      technology: SIP,
      type: URI,
      pattern: [ddd]@shongo.cesnet.cz,
    }
  ]
}
```

2.2 Reservation requests

Each controller runs a service component which processes incoming reservation requests which can be created by users through a controller client and the requests are persisted in the controller's storage. New reservation requests can be created and existing reservation requests can be modified or deleted. Class diagram for reservation requests is shown in fig. 2.2.

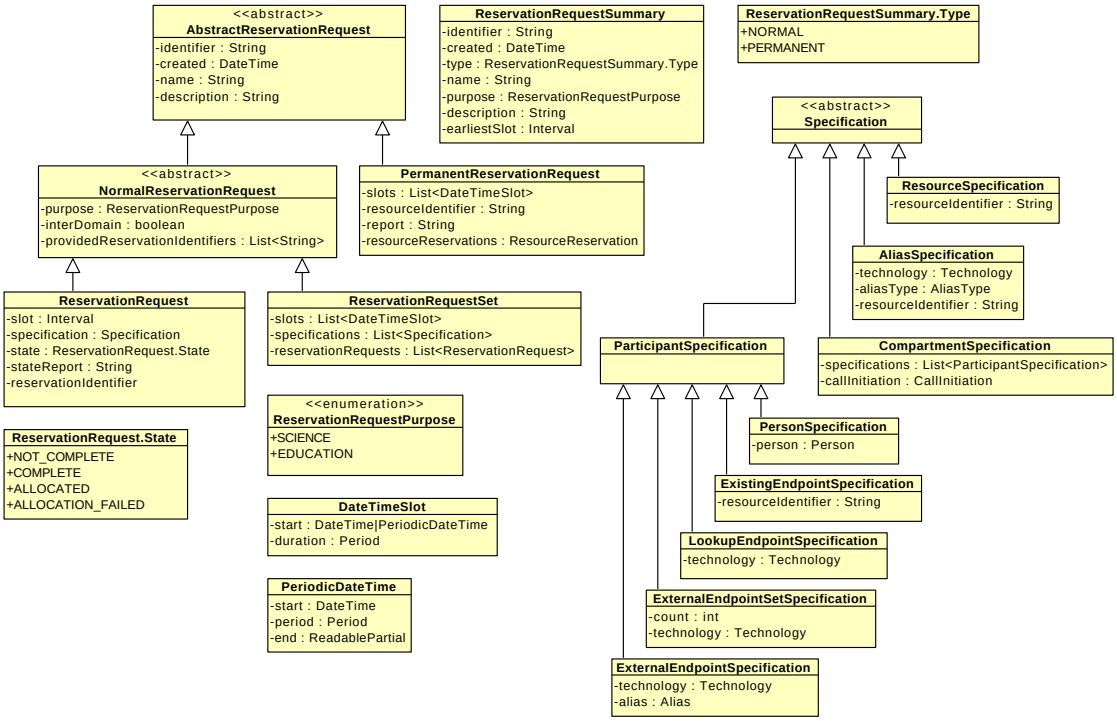

Figure 2.2: Class diagram for reservation requests in the controller API

There are 3 types of reservation requests:

1. **ReservationRequestSet** represents a compound request which can result into multiple **ReservationRequest**s. It must specify one or multiple date/time **slots** and one or multiple **specifications** for targets which the user wants to get allocated. A single date/time slot can be specified as periodic (e.g., each Tuesday from 14:00 to 16:00 from September to June).

   Multiple date/time slots allows us to specify multiple specific date/times (e.g., today and tomorrow from 13:00 to 15:00) or even multiple periodic date/times (e.g., each Tuesday from 14:00 to 16:00 and Thursday from 16:00 to 18:00 from September to June).

   Multiple specifications allows us to request multiple compartments or resources (e.g., we specify one compartment of H.323 and SIP participants for audio and video and another compartment in web conferencing technology to share content and these two compartments won't be interconnected but will be present at the same date/time slot or slots).

   Periodic date/time slots must be enumerated to absolute date/time slots and for each absolute date/time slot and each specification is created one **ReservationRequest** (a Cartesian product of enumerated **slots** and **specifications**).

   TODO: ReservationRequests created for the same date/time slot should be dependent to each other (allocation succeeds for all or none). [not implemented yet]

2. **ReservationRequest** represents a minimum processable unit for a scheduler which can be allocated to a reservation. It must specify exactly one absolute date/time **slot** (periodic date/time cannot be used here) and one **specification** for a target which should be allocated.

3. **PermanentReservationRequest** represents a special type of request specifying single resource which should be considered as not-allocatable in specified date/time slots. The request can be created by the resource owner. It is useful for disabling some date/time slots when the provided resource cannot be used by the scheduler.

Each reservation request is identified by an unique **identifier** and each request also has some basic attributes, e.g., date/time when the request was **created** or **name** of the request).

There are several types of specifications for targets which can be requested for allocation in reservation requests:

1. **ResourceSpecification** allows us to get allocated a specific resource. Then we can use it in the requested date/time slot for our purposes (e.g., we request a lecture room and we get allocated reservation for the the lecture room and we can do a presentation there).

2. **AliasSpecification** allows us to get allocated an available alias. When we receive an alias reservation we can use it for our purposes (e.g., assign the alias to our hardware endpoint or installed software client) or we can use it when creating another reservation request and pass the alias reservation to the request. A scheduler then can use the alias as available resource when it is allocating a reservation for the reservation request (e.g., assign it to any managed endpoint or created virtual room).

3. **ExistingEndpointSpecification** allows us to get allocated a specific endpoint device resource. Then we can use it in the requested date/time slot for our purposes (e.g., we request an H.323 endpoint in a lecture room and we get allocated reservation for the endpoint and also for the lecture room, we can go to the lecture room and use the endpoint to connect to a conference).

4. **LookupEndpointSpecification** allows us to get allocated an endpoint according to given parameters (e.g., technology of the endpoint). The scheduler will search for an available endpoint in the resource database and allocate it for us.

5. **CompartmentSpecification** allows us to get allocated a whole compartment. We specify one or multiple **participants** and the scheduler allocates for us all specified endpoints and finds available virtual rooms and aliases which are needed to establish the compartment and allocates them too. As part of the **CompartmentSpecification** we can also specify **callInitiation** – who should initiate the video conference call whether a virtual room or an endpoint (see **CallInitiation** for more details).

6. **ExternalEndpointSetSpecification** can be specified only inside a **CompartmentSpecification** and it allows us to request a compartment where should be available ports for external endpoints (e.g., user creates a reservation request for a compartment and he specifies that he want 3 external endpoints in H.323 and thus the scheduler allocates a reservation for a single H.323 virtual room for 3 ports; if the user has also specified one existing endpoint the scheduler would allocate a H.323 virtual room for 4 ports).

7. **ExternalEndpointSpecification** can be also specified only inside a **CompartmentSpecification** and it allows us to request a compartment where should be available port for external endpoint which can be also *call-out* from a virtual room (if the specification has **alias** specified).

8. **PersonSpecification** can be also specified only inside a **CompartmentSpecification** and it allows us to request a specific person to participate in a compartment.

   TODO: User must be invited to a videoconference (e.g., by email) and he must select an endpoint by which he will connect to the compartment. [not fully implemented yet]

A reservation request (**ReservationRequest**) can be in one of a several states (see fig. 2.3):

1. **NOT_COMPLETE** A reservation request is not ready for allocation by a scheduler (e.g., some person requested by the **PersonSpecification** hasn't confirmed/rejected the invitation or he hasn't selected an endpoint by which he will connect to a compartment).

2. **COMPLETE** A reservation request is ready for allocation by a scheduler, but the scheduler hasn't allocated it yet.

3. **ALLOCATED** A reservation request has been successfully allocated by the scheduler to a reservation.

4. **ALLOCATION_FAILED** The scheduler has failed to allocate a reservation request to a reservation.

Only complete reservation requests are allocated by the scheduler to reservations.
Here are some examples of reservation requests:

- Permanent reservation request which forbids the allocation of Tandberg Codec C90 endpoint in January 2012.

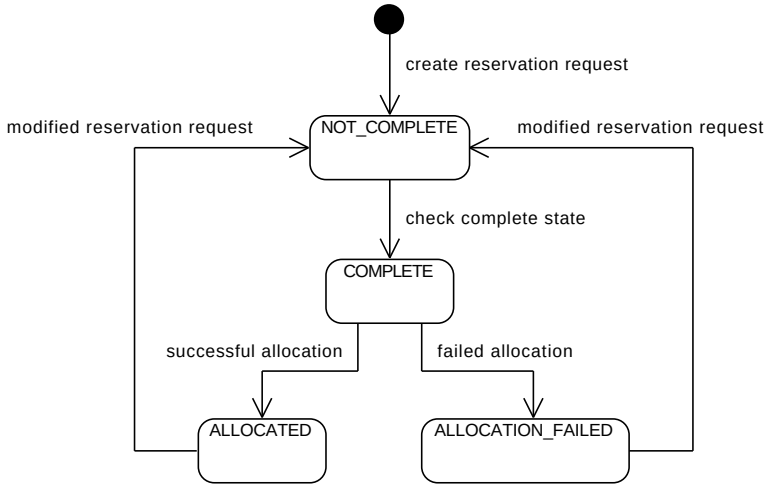

Figure 2.3: State diagram for reservation request in the controller API

```
request1 = PermanentReservationRequest {
  identifier: shongo:cz.cesnet:1,
  name: Example 1,
  slots: [
    DateTimeSlot {
      start: DateTime(2012-01-01T00:00),
      duration: Period(P1M)
    }
  ],
  resourceIdentifier: shongo:cz.cesnet:2 // Tandberg Codec C90
}
```

- Reservation request for a compartment taking place on 12.10.2012 from 14:00 to 16:00 in which will participate Tandberg Codec C90 endpoint from the resource database (`shongo:cz.cesnet:2`) and to the compartment can also connect any other three H.323 endpoints. Thus for the compartment must be allocated one H.323 virtual room with 4 available ports.

```
request2 = ReservationRequest {
  identifier: shongo:cz.cesnet:2,
  name: Example 2,
  slot: DateTimeSlot {
    start: DateTime(2012-10-12T14:00),
    duration: Period(PT2H)
  },
  specification: CompartmentSpecification {
    specifications: [
      ExternalEndpointSetSpecification {
        count: 3,
        technology: H323
      },
      ExistingEndpointSpecification { // Tandberg Codec C90
        resourceIdentifier: shongo:cz.cesnet:2
      }
    ]
  }
}
```

- Set of reservation requests from which will be created a reservation request for each Wednesday from 14:00 to 16:00, first on 5.9.2012 and last on 26.6.2013. Each created request will be allocated to a compartment reservation with a virtual room for 3 ports in H.323.

```
request3 = ReservationRequestSet {
  identifier: shongo:cz.cesnet:3,
  name: Example 3,
  slots: [
    DateTimeSlot {
      start: PeriodicDateTime{
        start: DateTime(2012-09-05T14:00),
        period: Period(PT2H),
        end: ReadablePartial(2013-06-31)
      },
      duration: Period(PT2H)
    }
  ],
  specifications: [
    CompartmentSpecification {
      specifications: [
        ExternalEndpointSetSpecification {
          count: 3,
          technology: H323
        }
      ]
    }
  ]
}
```

- Reservation request for a H.323 phone number from 1.1.2012 to 31.12.2012.

```
request4 = ReservationRequest {
  identifier: shongo:cz.cesnet:4,
  name: Example 4,
  slot: DateTimeSlot {
    start: DateTime(2012-01-01T00:00),
    duration: Period(PT1Y)
  },
  specification: AliasSpecification {
    technology: H323,
    aliasType: E164
  }
}
```

2.3 Reservations

Reservation is an object through which an user receives allocated resources which he was requesting by a reservation request. Each reservation is identified by an unique **identifier** and the reservation is allocated for one specific date/time **slot**. Reservation can contain some child reservations which has been allocated to satisfy parent allocation needs (e.g., a reservation for a compartment may contain child reservations for endpoints, virtual rooms or aliases, and a reservation for an endpoint may contain child reservation for a lecture room).

There are several types of reservations:

1. **ResourceReservation** represents a reservation for a resource from the resource database (e.g., reservation for a H.323 endpoint for a specific date/time slot or a reservation for a

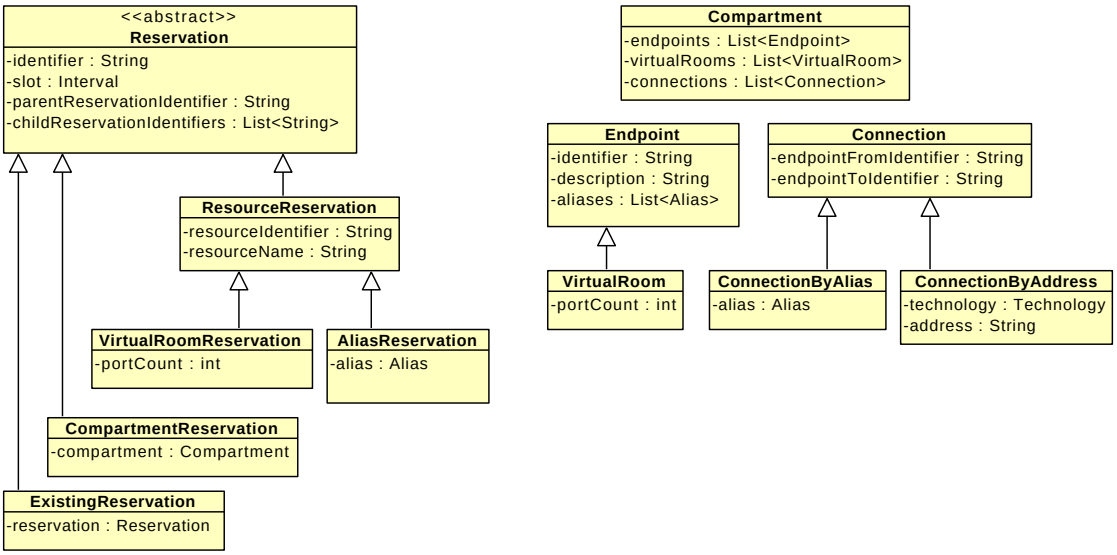

Figure 2.4: Class diagram for reservations in the controller API

lecture room for a specific date/time slot). For each resource and date/time slot can exist only one reservation (reservation must not collide).

2. **`VirtualRoomReservation`** represents a special type of **`ResourceReservation`** for a virtual room in the device resource with a **`VirtualRoomsCapability`** (e.g., a reservation for a virtual room with 5 available ports in H.323 MCU resource). These reservations can collide (multiple virtual rooms can be created at the same time), but must not exceed the maximum number of ports specified in the **`VirtualRoomsCapability`**.

3. **`AliasReservation`** represents a special type of **`ResourceReservation`** for an **`alias`** from the resource with an **`AliasProviderCapability`** (e.g., a reservation for an alias `950087201` from a resource which provides aliases with prefix `9500872XX`).

4. **`CompartmentReservation`** represents a reservation for a compartment and it usually contains multiple child reservations for virtual rooms, endpoints and aliases. It also contains a reference to **`compartment`** (instance of **`Compartment`** class) which represents a compartment plan which is used by an executor to establish the compartment.

5. **`ExistingReservation`** represents a reused reservation (which was provided in the reservation request).

The following two diagrams describes how **`ReservationRequest`**s and **`ReservationRequestSet`**s are processed by a controller and allocated to **`Reservation`**s (fig. 2.5 and 2.6). Each controller runs these components:

1. **Reservation service** accepts new reservation requests from users (through controller clients) and also new reservations from the scheduler and persist them to controller's storage.

2. **Preprocessor** uses the service to get not-preprocessed **`ReservationRequestSet`**s and creates new **`ReservationRequest`**s from them. Preprocessor creates only requests which take place in

working interval (e.g., one month ahead). It allows an user to create a **`ReservationRequestSet`** specifying never-ending periodic date/time (e.g., each Thursday from 14:00 to 16:00) and as the weeks goes the preprocessor will create more and more **`ReservationRequest`**s.

3. **Scheduler** uses the service to get complete **`ReservationRequest`**s and allocates them to **`Reservation`**s. The scheduler also allocates only requests in a working interval (e.g., one month ahead).

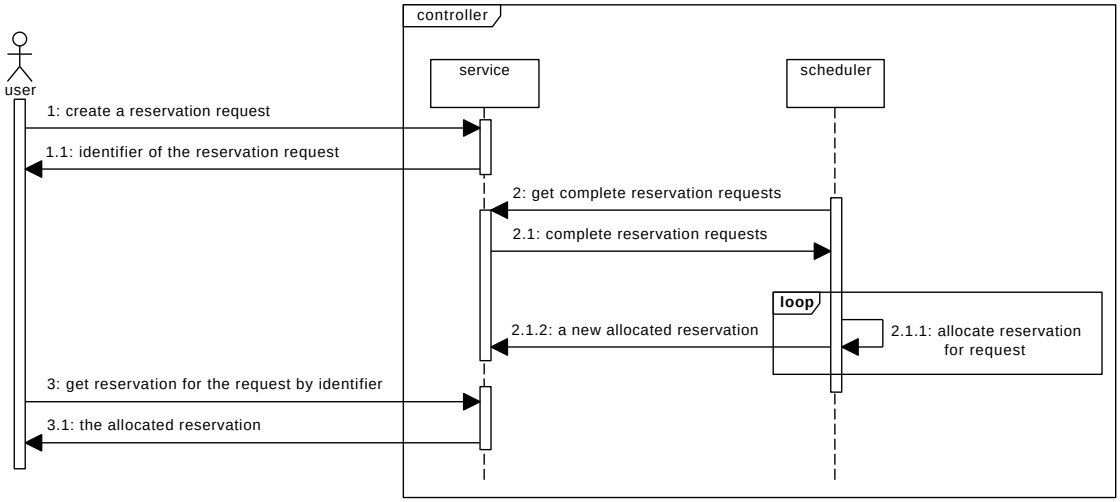

Figure 2.5: Interaction diagram depicting allocation of reservation for a single reservation request

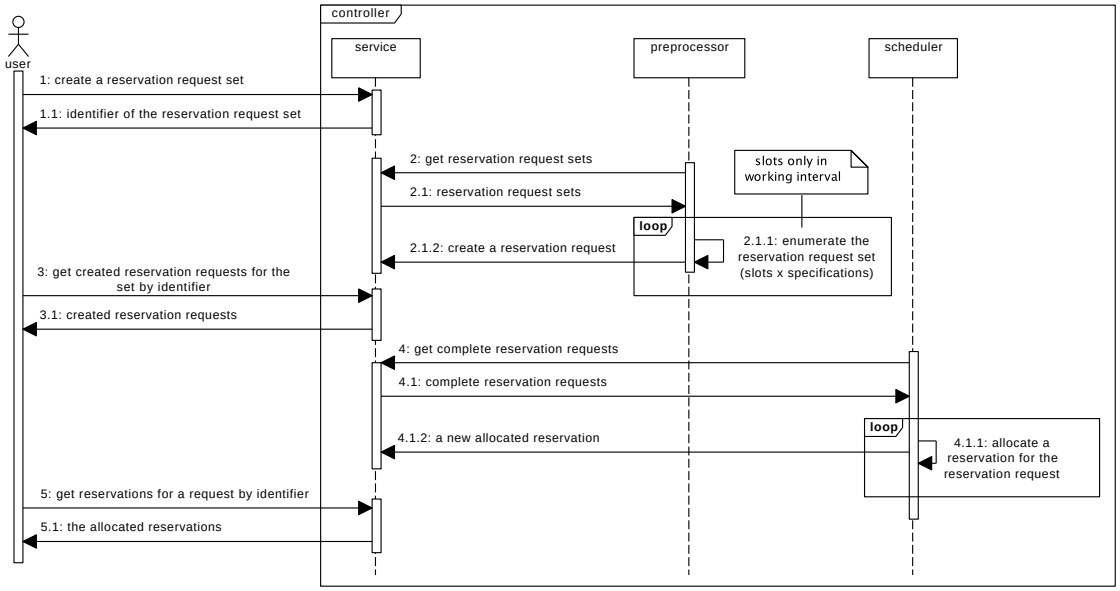

Figure 2.6: Interaction diagram depicting allocation of reservations for set of reservation requests

Here are some examples of allocated reservations:

- Allocated permanent reservation for `request1` (which requested forbidding allocation of Tandberg endpoint) that was described in section 2.2.

```
reservation1 = ResourceReservation {
  identifier: shongo:cz.cesnet:1,
  slot: Interval(2012-01-01T00:00/2012-02-01T00:00),
  resourceIdentifier: shongo:cz.cesnet:2
}
```

- Allocated reservation for `request2` (which was requesting compartment for 3 external H.323 ports and existing H.323 endpoint) that was described in section 2.2.

```
reservation2 = CompartmentReservation {
  identifier: shongo:cz.cesnet:2,
  slot: Interval(2012-10-12T14:00/2012-10-12T16:00),
  childReservationIdentifiers: [
    shongo:cz.cesnet:3, // Child reservation for Tandberg endpoint
    shongo:cz.cesnet:4, // Child reservation for virtual room in Codian MCU
    shongo:cz.cesnet:5  // Child reservation for alias for virtual room
  ],
  compartment: Compartment { // Executor compartment plan
    endpoints: [
      Endpoint { // Tandberg Codec C90 endpoint
        identifier: 2,
        description: existing endpoint shongo:cz.cesnet:2,
      }
    ],
    virtualRooms: [
      VirtualRoom { // Virtual room with 4 available ports in Codian MCU 4515
        identifier: 3,
        portCount: 4,
        description: virtual room in device resource shongo:cz.cesnet:1,
        aliases: [
          Alias { technology: H323, type: E164, value: 950087201 }
        ]
      }
    ],
    connections: [
      ConnectionByAlias { // Connection from Tandberg endpoint to virtual room
        endpointFromIdentifier: 2,
        endpointToIdentifier: 3,
        alias: Alias { technology: H323, type: E164, value: 950087201 }
      }
    ]
  }
}

reservation3 = ResourceReservation { // Child reservation for Tandberg endpoint
  identifier: shongo:cz.cesnet:3,
  slot: Interval(2012-10-12T14:00/2012-10-12T16:00),
  parentReservationIdentifier: shongo:cz.cesnet:2,
  resourceIdentifier: shongo:cz.cesnet:2
}
```

```
reservation4 = VirtualRoomReservation { // Child reservation for virtual room in Codian MCU
  identifier: shongo:cz.cesnet:4,
  slot: Interval(2012-10-12T14:00/2012-10-12T16:00),
  parentReservationIdentifier: shongo:cz.cesnet:2,
  resourceIdentifier: shongo:cz.cesnet:1,
  portCount: 4
}

reservation5 = AliasReservation { // Child reservation for alias for virtual room
  identifier: shongo:cz.cesnet:5,
  slot: Interval(2012-10-12T14:00/2012-10-12T16:00),
  parentReservationIdentifier: shongo:cz.cesnet:2,
  resourceIdentifier: shongo:cz.cesnet:1,
  alias: Alias { technology: H323, type: E164, value: 950087201 }
}
```

- Allocated reservation for `request3` on 5.9.2012 (which was requesting compartment for 3 external H.323 ports for multiple date/time slots) that was described in section 2.2.

```
reservation6 = CompartmentReservation {
  identifier: shongo:cz.cesnet:6,
  slot: Interval(2012-09-05T14:00/2012-09-05T16:00),
  childReservationIdentifiers: [
    shongo:cz.cesnet:7, // Child reservation for virtual room in Codian MCU
    shongo:cz.cesnet:8  // Child reservation for alias for virtual room
  ],
  compartment: Compartment { // Executor compartment plan
    virtualRooms: [
      VirtualRoom { // Virtual room with 3 available ports in Codian MCU 4515
        identifier: 3,
        portCount: 4,
        description: virtual room in device resource shongo:cz.cesnet:1,
        aliases: [
          Alias { technology: H323, type: E164, value: 950087201 }
        ]
      }
    ]
  }
}

reservation7 = VirtualRoomReservation { // Child reservation for virtual room in Codian MCU
  identifier: shongo:cz.cesnet:7,
  slot: Interval(2012-09-05T14:00/2012-09-05T16:00),
  parentReservationIdentifier: shongo:cz.cesnet:6,
  resourceIdentifier: shongo:cz.cesnet:1,
  portCount: 3
}

reservation8 = AliasReservation { // Child reservation for alias for virtual room
  identifier: shongo:cz.cesnet:8,
  slot: Interval(2012-09-05T14:00/2012-09-05T16:00),
  parentReservationIdentifier: shongo:cz.cesnet:6,
  resourceIdentifier: shongo:cz.cesnet:1,
  alias: Alias { technology: H323, type: E164, value: 950087201 }
}
```

- Allocated reservation for `request4` (which was requesting H.323 phone number) that was described in section 2.2.

```
reservation9 = AliasReservation {
  identifier: shongo:cz.cesnet:9,
  slot: Interval(2012-01-01T00:00/2013-01-01T00:00),
  resourceIdentifier: shongo:cz.cesnet:5, // From aliasProvider resource
  alias: Alias {
    technology: H323,
    type: E164,
    value: 950087202
  }
}
```

TODO: Describe that scheduler can in future use inter-controller protocol to allocate resources for reservations

2.4 Authentication & Authorization

2.4.1 Controller vs. Client

Controller clients communicate with a controller through XML-RPC [2]. When a controller client sends a XML-RPC request to a controller and the request should specify an user identity who sent it (referred to as *secure request*), the request must contain **`SecurityToken`** as the first parameter. **`SecurityToken`** equals to OpenID [3] access token.

Before controller client is able to send secure request, it must first obtain an access token for the current user. According to OpenID specification the access token can be obtained from the authorization server as described in fig. 2.7.

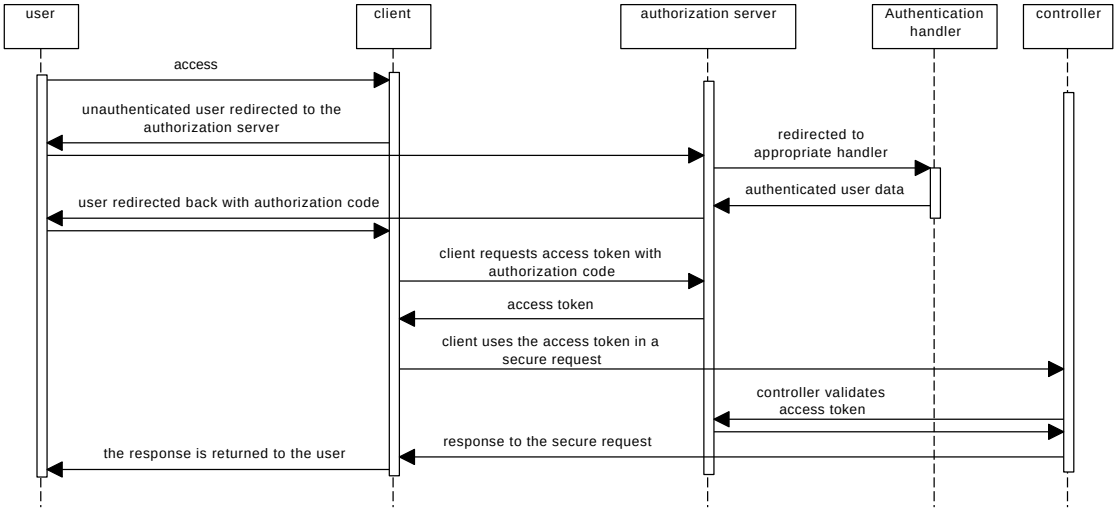

Figure 2.7: Diagram depicting user authentication through controller client to controller

TODO: Describe controller API authorization

2.4.2 Controller vs. Connector

TODO: Describe connector API authentication

Chapter 3

Common Data Types

In this chapter common atomic types, enum types and object classes for Shongo API are described.

3.1 Failure Related

Each failure in API is reported by **fault code** (number value) and **fault string** (description text). List of common faults:

fault code	description	fault string
0	Unknown fault.	Describes the unknown fault.
10	The `class` is not defined.	Specifies which `class`.
11	The `class` cannot be instanced.	Specifies which `class`.
12	The attribute is not defined.	Specifies which `attribute` in which `class`.
13	The attribute type is wrong.	Specifies which `attribute` in which `class` and also it specifies the present and required type.
14	The attribute wasn't present and is required.	Specifies which `attribute` in which `class`.
15	The collection is empty and is required.	Specifies which `collection` in which `class`.
16	The attribute was present but is read-only.	Specifies which `attribute` in which `class`.
17	Wrong value for attribute was present.	Specifies which `value` in which `attribute` in which `class`.
20	The value of an enum attribute is wrong.	Specifies which `value`.
21	Failed to parse date/time value.	Specifies which date/time `value`.
22	Failed to parse period value.	Specifies which period `value`.

23	Failed to parse interval value.	Specifies which interval `value`.
24	Failed to parse partial date/time value.	Specifies which partial date/time `value`.
30	Item with wrong type was present in collection.	Specifies which `item` in which `collection` and types which are allowed.
31	Null value cannot be used as item in collection.	Specifies which `collection`.
40	Entity was not found	Specifies which `entity`.
41	Entity validation failed	Specifies which `entity`.
99	The request is not implemented yet. An application log must be checked for more details.	Specifies what is not implemented.

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.

3.2 Security and Identity Related

- **`atomic_type SecurityToken = String`**

  Represents an identity and credentials of a user performing a request. **`SecurityToken`** equals to OpenID [3] access token (see section 2.4 for more details).

- **`class Person`**

  Represents a person that can access a Shongo videoconference.

  Attributes:

  - **`String name`** (**Required**)
    Name to be displayed.
  - **`String email`** (**Required**)
    Email to which a videoconference invitation should be sent.

3.3 Time Related

- **`atomic_type Period = String`**

  Used for representing period or duration. Format is specified by 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*).

  Example: We want to set videoconference duration:

  ```
  duration = PT2H
  ```

- **`atomic_type DateTime = String`**

  Used for representing an absolute date/time. Format is specified by ISO8601 date/time in UTC (e.g., 20120130T10:09:55) or with explicit timezone specification (e.g., 20120130T10:09:55+02:00).

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

  ```
  dateTime = 2012-12-31T12:00
  ```

- **`atomic_type RelativeDateTime = Period`**

  Used for representing a relative date/time. Relative date/time can be evaluated to absolute date/time by specifying a referenced absolute date/time.

  Example: We want to define a maximum future to which a resource can be scheduled. We can specify it by **`RelativeDateTime`** as follows:

  ```
  relativeDateTime = P4M
  ```

  The `relativeDateTime` can be evaluated by using reference date/time `2012-01-01T00:00` to `2012-05-01T00:00`.

- **`class PeriodicDateTime`**

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

  Attributes:

  - **`DateTime 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.
  - **`DateTime end`** (**Optional**)
    Ending date/time for events to not occur forever (not required, we can specify infinite periodic date/time).
  - **`List<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 **`PeriodicDateTime`** 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 }
    }
```

- **atomic_type Interval = String**

  Represents the time between two instants. Values must be in format <start>/<duration> where <start> is in **DateTime** format and <duration> in **Period** format.

3.4 Other

- **enum Technology**

  Enumeration of supported (or planned to be supported) videoconference technologies.

  Enumeration values:

  - **H323**
  - **SIP**
  - **ADOBE_CONNECT**
  - **SKYPE**
  - **BIG_BLUE_BUTTON**
  - **OPEN_MEETING**
  - **WEBEX**

- **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).
  - **DateTime 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)
  - **RoomLayout layout** (**Optional**)
    User layout, overriding the room default layout.

Chapter 4

Controller API

4.1 Communication Protocol

Controller API is provided through XML-RPC [2]. In XML-RPC all described types are represented as follows:

- Values of atomic types are represented by their's equivalent in XML-RPC (e.g., integer value `42` as `<i4>42</i4>` or string value `Hello` as `<string>Hello</string>`).
- Enum values are represented as strings (e.g., `ReservationRequestType.PERMANENT` as `<string>PERMANENT</string>`).
- Arrays and collections are represented as XML-RPC arrays, e.g.

  ```
  [101, 202]
  ```

  as

  ```
  <array>
    <data>
      <value><i4>101</i4></value>
      <value><i4>202</i4></value>
    </data>
  </array>
  ```

- Object instances are represented as XML-RPC struct types with special `class` attribute specifying object type, e.g.

  ```
  Person {
    name: "Martin Srom",
    email: "srom@cesnet.cz"
  }
  ```

  as

```
<struct>
  <member>
    <name>class</name>
    <value><string>Person</string></value>
  </member>
  <member>
    <name>name</name>
    <value><string>Martin Srom</string></value>
  </member>
  <member>
    <name>email</name>
    <value><string>srom@cesnet.cz</string></value>
  </member>
</struct>
```

- Null values are represented as empty XML-RPC struct type (<struct></struct>). It is useful e.g., when the user want to clear attribute value by any modify API method. He should set the attribute value to empty struct and the value will be cleared on the server.
- Failures are propagated through XML-RPC by **`faultCode`** and **`faultString`** values.

4.2 Data types

This section contains description of types used only in Controller API.

- **`class Controller`**

  Descriptor of a controller.

  Attributes:

  - **`Domain domain`** (**ReadOnly**)
    Domain which is controlled by the controller.

- **`class Domain`**

  Descriptor of a domain.

  Attributes:

  - **`String name`** (**Required**)
    A unique domain name (e.g., `cz.cesnet`).
  - **`String organization`** (**Optional**)
    Name of organization owning the domain (e.g., `CESNET, z.s.p.o.`).
  - **`Status status`** (**Required**)
    Status whether the domain controller is available.

- **`class Connector`**

  Descriptor of a connector agent which is managing a device resource.

  Attributes:

  - **`String name`** (**ReadOnly**)
    Unique name of the connector agent within the domain (name of JADE agent).
  - **`String resourceIdentifier`** (**ReadOnly**)

Identifier of a device resource which is managed by the connector agent.
  - **`Status status`** (**Required**)
    Status whether the connector agent is available to the controller.

- **`enum Status`**

  Status of a **`Domain`** or **`Connector`**.

  Enumeration values:

  - **`AVAILABLE`**
    Means that a **`Domain`** or **`Connector`** is available to the controller.
  - **`NOT_AVAILABLE`**
    Means that a **`Domain`** or **`Connector`** is not currently available to the controller.

- **`class DateTimeSlot`**

  Date/time slot can represent one or more **`Interval`** values. Definition of date/time slot is a pair of starting date/time and duration, where starting date/time can be periodic and thus can result into multiple absolute date/times.

  Attributes:

  - **`DateTime|PeriodicDateTime start`** (**Required**)
    Defines the start of date/time slot (or multiple starts in case of periodic date/time).
  - **`Period duration`** (**Required**)
    Defines the duration of date/time slot.

  For reservation purposes, the collection of **`DateTimeSlot`**s 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 colllection of **`Interval`**s.

- **`class Alias`**

  Represents an alias for a device.

  Attributes:

  - **`Technology technology`** (**Required**)
    Technology of the alias (e.g., `H323`).
  - **`AliasType type`** (**Required**)
    Type of the alias (e.g., `E164`).
  - **`String value`** (**Required**)
    Value of the alias (e.g., `950087704`).

- **`enum AliasType`**

  Enumeration of available types of aliases.

  Enumeration values:

  - **`E164`**
    See `http://en.wikipedia.org/wiki/E.164`.
  - **`IDENTIFIER`**
    e.g., H.323 ID.
  - **`URI`**
    e.g., SIP URI.

- **`enum CallInitiation`**

  Enumeration of preference who should initiate a video conference call.

  Enumeration values:

  - **`TERMINAL`**
    A terminal should initiate the video conference call to a virtual room (it is referred to as *call in*).
  - **`VIRTUAL_ROOM`**
    A virtual room should initiate the video conference call to a terminal (it is referred to as *call out*).

- **`class Resource`**

  This class represents a complete resource definition. This class is used for creating and modifying resources.

  Attributes:

  - **`String identifier`** (**Required**)
    Resource unique identifier as defined in UC-4.
  - **`String parentIdentifier`** (**Optional**)
    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.
  - **`List<Capability> capabilities`** (**Optional**)
    List of capabilities which the resource has.
  - **`String description`** (**Optional**)
    Long description depicting the resource.
  - **`boolean allocatable`** (**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 **`allocatable`** to `false` and restrict the time when the resource can by used for public scheduling (e.g., setup permanent reservations) and then modify the **`allocatable`** to `true`.
  - **`DateTime maxFuture`** (**Optional**)
    The maximum future time for reservations as defined in UC-12.
  - **`List<String> childResourceIdentifiers`** (**Optional**)
    List of child resources identifiers (e.g., the resource can be a physical room and the collection **`childResourceIdentifiers`** contains all endpoints in the room).

- **`class DeviceResource extends Resource`**

  Represents a complete device resource definition.

  Attributes:

  - **`String address`** (**Optional**)
    Address of the device resource (e.g., `147.251.1.1` or `connect.cesnet.cz`).
  - **`Set<Technology> technologies`** (**Required**)
    Set of technologies which are supported by the device resource.
  - **`String |ManagedMode mode`** (**Optional**)
    String value `UNMANAGED` specifies that the device resource is not managed by any connector agent. **`ManagedMode`** specifies by which connector agent is the device resource managed.

- **class** `ManagedMode`

  Represents a descriptor of connector agent by which a device resource can be managed.

  Attributes:

  - `String connectorAgentName` (Optional)
    Name of the connector agent which is managing the device resource.

- **class** `Capability`

  Base class for all capabilities which a resource can have. A resource can have zero, one or multiple capabilities.

- **class** `VirtualRoomsCapability` **extends** `Capability`

  Capability for devices which provides virtual rooms for interconnecting multiple other endpoints (e.g., device resource for H.323 MCU will have the `VirtualRoomsCapability`).

  Attributes:

  - `int portCount` (Required)
    Maximum number of ports which can be allocated for all virtual rooms at one moment.

- **class** `TerminalCapability` **extends** `Capability`

  Capability for devices which can participate in a compartment.

- **class** `StandaloneTerminalCapability` **extends** `TerminalCapability`

  Capability for devices which can participate in a compartment or which can connect to another device (even without a virtual room). (e.g., the device resource for a H.323 endpoint will have the `StandaloneTerminalCapability` or the device resource for an Adobe Connect client will have the `TerminalCapability`).

- **class** `AliasProviderCapability` **extends** `Capability`

  Capability provides that the resource can be allocated as a alias.

  Attributes:

  - `Technology technology` (Required)
    Technology for which the alias will be allocated.
  - `AliasType type` (Required)
    Type of alias which will be allocated.
  - `String pattern` (Required)
    Pattern specifying which values will be generated as alias (e.g., `950087[ddd]` pattern for generation of `950087001`, `950087002`, ... `950087999` values).
  - `boolean restrictedToOwnerResource` (Required)
    Specifies whether aliases can be allocated only for the resource which has this capability or for all other device resources.

- **class** `ResourceSummary`

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

  Attributes:

  - `String identifier` (ReadOnly)

Resource unique identifier.
  - `String name` (**ReadOnly**)
    Resource name which can be displayed.
  - `String technologies` (**ReadOnly**)
    Comma separated list of supported technologies.
  - `String parentIdentifier` (**Optional**)
    A parent resource identifier in which is the resource located.

- `enum ReservationRequestPurpose`

  Enumeration values:

  - `SCIENCE`
    Reservation is requested for research purposes.
  - `EDUCATION`
    Reservation is requested for education purposes (e.g., for a lecture).

- `class AbstractReservationRequest`

  Represents a base class for all possible types reservation requests. It contains only common attributes which are same for all types of reservation requests.

  Attributes:

  - `String identifier` (**ReadOnly**)
    Reservation request unique identifier as defined in UC-7.
  - `DateTime created` (**ReadOnly**)
    Date/time when the reservation request was created.
  - `String name` (**Required**)
    Name of the reservation request.
  - `String description` (**Optional**)
    Detailed reservation request description.

- `class PermanentReservationRequest`

  Represents a permanent reservation request as defined in UC-10.

  Attributes:

  - `List<DateTimeSlot> slots` (**Required**)
    List of requested `slots`.
  - `String resourceIdentifier` (**Required**)
    Identifier of resource for which the permanent reservation(s) should be created.
  - `String report` (**ReadOnly**)
    Description of current state of permanent reservation request (can contain reasons, e.g., why the allocation of any reservations failed).
  - `List<ResourceReservation> resourceReservations` (**ReadOnly**)
    List of reservations which was already allocated for the resource.

- `class NormalReservationRequest extends AbstractReservationRequest`

  Represents an one time or periodic reservation request as defined in UC-8 and UC-9 (one time reservation request is a special case of periodic reservation request).

  Attributes:

  - `ReservationRequestPurpose purpose` (**Required**)

Purpose of the reservation request, see **ReservationRequestPurpose**.
  - **boolean interDomain** (**Optional**, default: false)
    Specify whether the scheduler should try allocate also resources from other domains.
  - **List<String> providedReservationIdentifiers** (**Optional**)
    List of identifiers for reservations which are provided to the reservation request and the scheduler can use all allocated resources from these reservations when allocating the reservation request.

- **class ReservationRequestSet extends NormalReservationRequest**

  Represents a reservation request for one or multiple reservations. It specifies one or multiple date/time slots and one or multiple specifications and for each combination (slots x specifications) will be created one **ReservationRequest**.

  Attributes:

  - **List<DateTimeSlot> slots** (**Required**)
    List of requested **slots**.
  - **List<Specification> specifications** (**Required**)
    List of requested **specifications**.
  - **List<ReservationRequest> reservationRequests** (**Required**)
    List of already created **ReservationRequest**s.

- **class ReservationRequest extends NormalReservationRequest**

  Represents a reservation request for single specification at one specific date/time slot. Only **ReservationRequest**s are processed by the scheduler and the scheduler allocates them to reservations.

  Attributes:

  - **Interval slot** (**Required**)
    Requested date/time **slot**.
  - **Specification specification** (**Required**)
    Requested **specification** for a target which should be allocated.
  - **ReservationRequest.State state** (**ReadOnly**)
    Current state of the reservation request.
  - **String stateReport** (**ReadOnly**)
    Description for the current state (can contain reason, e.g., why the allocation failed).
  - **String reservationIdentifier** (**ReadOnly**)
    Identifier of allocated reservation for this reservation request.

- **enum ReservationRequest.State**

  Enumeration of possible states in which a reservation request can be.

  Enumeration values:

  - **NOT_COMPLETE**
    A specification in the reservation request need to be additionally filled.
  - **NOT_ALLOCATED**
    The reservation request was not processed by the scheduler yet.
  - **ALLOCATED**
    The reservation request was successfully allocated to a reservation.
  - **ALLOCATION_FAILED**
    Allocation of the reservation request has failed.

- `class Specification`

  Represents a base class for all possible specifications. Each specification describes a target which is requested by a reservation request.

- `class ResourceSpecification extends Specification`

  Represents a specification for an resource which is stored in local or foreign domain controller and thus it has assigned a resource identifier.

  Attributes:

  - `String resourceIdentifier` (**Required**)
    Resource identifier for the resource.

- `class CompartmentSpecification extends Specification`

  Represents a specification for a group of participants which should participate a single compartment.

  Attributes:

  - `List<ParticipantSpecification> specifications` (**Required**)
    List of specifications for participants which should participate in a compartment.
  - `CallInitiation callInitiation` (**Optional**)
    Default `CallInitiation` in the compartment.

- `class ParticipantSpecification extends Specification`

  Represents a base class for specifications which can be added to the `CompartmentSpecification`.

- `class ExternalEndpointSpecification extends ParticipantSpecification`

  Represents a specification for one or multiple external endpoints. An external endpoint doesn't have assigned a resource identifier.

  Attributes:

  - `Technology technology` (**Required**)
    Technology of the external endpoint(s).
  - `int count` (**Optional**)
    Number of the external endpoint (all have the same technology).

- `class ExistingEndpointSpecification extends ParticipantSpecification`

  Represents a specification for an endpoint which is stored in local or foreign domain controller and thus it has assigned a resource identifier.

  Attributes:

  - `String resourceIdentifier` (**Required**)
    Device resource identifier for the endpoint.

- `class LookupEndpointSpecification extends ParticipantSpecification`

  Represents a specification of parameters for an endpoint which will be used to lookup a matching endpoint in local or foreign domain controller.

  Attributes:

  - `Technology technology` (**Required**)
    Technology of the requested endpoint.

- **class** `AliasSpecification` **extends** `Specification`

  Represents a specification for an alias.

  Attributes:

  - `Technology technology` (Optional)
    Technology of the requested alias.
  - `AliasType aliasType` (Optional)
    Type of the requested alias.
  - `String resourceIdentifier` (Optional)
    Identifier of a resource with `AliasProviderCapability` which is preferred to be used as the alias provider.

- **class** `PersonSpecification` **extends** `Specification`

  Represents a specification for a person that should participate in a compartment.

  Attributes:

  - `Person person` (Required)
    Requested person.

- **enum** `ReservationRequestSummary.Type`

  Enumeration values:

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

- **class** `ReservationRequestSummary`

  This class represents a summary of a reservation request. The summary is lightweight and does not contain all reservation request attributes. It is suitable when listing a lot of reservation requests from the controller database where the detail information about reservation request is not appropriate.

  Attributes:

  - `String identifier` (ReadOnly)
    Reservation request unique identifier as defined in UC-7.
  - `ReservationRequestSummary.Type type` (ReadOnly)
    Type of the reservation request, see `ReservationRequestSummary.Type`.
  - `String name` (ReadOnly)
    Name of the reservation request.
  - `ReservationRequestPurpose purpose` (ReadOnly)
    Purpose of the reservation request, see `ReservationRequestPurpose`.
  - `String description` (ReadOnly)
    Description of reservation request.
  - `Interval earliestSlot` (ReadOnly)
    Specifies the first future date/time slot for which the reservation is requested.

- **class** `Reservation`

  Represents base class for all possible types of successfully allocated reservation(s) for a reservation request. It contains only read only data that are obtained from a scheduler.

Attributes:

- `String identifier` (**ReadOnly**)
  Reservation unique identifier as defined in UC-7.
- `Interval slot` (**ReadOnly**)
  Date/time slot for which the reservation is allocated.
- `String parentReservationIdentifier` (**ReadOnly**)
  Identifier of parent reservation.
- `List<String> childReservationIdentifiers` (**ReadOnly**)
  Identifiers of child reservations. Each reservation can contain multiple child reservations which have been allocated to satisfy the parent reservation needs.

- `class ResourceReservation extends Reservation`

  Represents a successfully allocated resource.

  Attributes:

  - `String resourceIdentifier` (**ReadOnly**)
    Allocated resource identifier.
  - `String resourceName` (**ReadOnly**)
    Allocated resource name.

- `class VirtualRoomReservation extends ResourceReservation`

  Represents a successfully allocated device resource as a virtual room.

  Attributes:

  - `String portCount` (**ReadOnly**)
    Number of allocated ports for the virtual room.

- `class AliasReservation extends ResourceReservation`

  Represents a successfully allocated resource as a alias.

  Attributes:

  - `Alias alias` (**ReadOnly**)
    Allocated alias.

- `class CompartmentReservation extends Reservation`

  Represents a successfully allocated compartment.

  Attributes:

  - `Compartment compartment` (**ReadOnly**)
    Allocated compartment.

- `class ExistingReservation extends Reservation`

  Represents a reused reservation (which was provided to the reservation request).

  Attributes:

  - `Reservation reservation` (**ReadOnly**)
    Existing reservation which is reused.

- `class Compartment`

  Represents an allocated compartment.

Attributes:

- `List<Compartment.Endpoint> endpoints` (**ReadOnly**)
  List of participating endpoints.
- `List<Compartment.VirtualRoom> virtualRooms` (**ReadOnly**)
  List of virtual rooms.
- `List<Compartment.Connection> connections` (**ReadOnly**)
  List of connection between endpoints and virtual rooms.

- `class Compartment.Endpoint`

  Represents an endpoint in allocated compartment.

  Attributes:

  - `String description` (**ReadOnly**)
    Description of the endpoint.
  - `List<Alias> aliases` (**ReadOnly**)
    List of assinged aliases to the endpoint.

- `class Compartment.VirtualRoom extends Compartment.Endpoint`

  Represents an virtual room in allocated compartment.

  Attributes:

  - `String portCount` (**ReadOnly**)
    Number of allocated ports for the virtual room.

- `class Compartment.Connection`

  Represents an connection between two endpoints in allocated compartment.

  Attributes:

  - `String endpointFrom` (**ReadOnly**)
    Description of source endpoint which initiates the call.
  - `String endpointTo` (**ReadOnly**)
    Description of target endpoint.

- `class Compartment.ConnectionByAlias extends Compartment.Connection`

  Represents an connection between two endpoints by alias in allocated compartment.

  Attributes:

  - `Alias alias` (**ReadOnly**)
    Alias which is used.

- `class Compartment.ConnectionByAddress extends Compartment.Connection`

  Represents an connection between two endpoints by address in allocated compartment.

  Attributes:

  - `Technology technology` (**ReadOnly**)
    Technology which is used.
  - `String address` (**ReadOnly**)
    Address which is used.

- **`class ResourceAllocation`**

  Represents an information about allocation of resource.

  Attributes:

  - **`String identifier`** (**ReadOnly**)
    Resource identifier.
  - **`String name`** (**ReadOnly**)
    Resource name.
  - **`Interval interval`** (**ReadOnly**)
    Interval for which the allocation information is contained.
  - **`List<ResourceReservation> reservations`** (**ReadOnly**)
    List of reservations for the resource.

- **`class VirtualRoomsResourceAllocation extends ResourceAllocation`**

  Represents an information about allocation of device resource with virtual rooms.

  Attributes:

  - **`int maximumPortCount`** (**ReadOnly**)
    Maximum port count which are available in the Device resource for all virtual rooms at one moment.
  - **`int availablePortCount`** (**ReadOnly**)
    Available port count which are available in the Device resource for **`interval`**.

4.3 Common

- **`Controller getController()`**

  Get information about the domain controller. See **`Controller`** class.

- **`Collection<Domain> listDomains(SecurityToken token)`**

  Lists all known domains with status if they are available to the domain controller.

- **`Collection<Connector> listConnectors(SecurityToken token)`**

  Lists all known connectors in the controlled domain.

4.4 Resources

- **`String createResource(SecurityToken token, Resource resource)`**

  Create a new resource that will be stored in the domain controller. The new resource identifier is returned as a result. The user with given **`token`** will be the resource owner. The **`resource`** must contain all attributes marked as **Required**.

- **`void modifyResource(SecurityToken token, Resource resource)`**

  Modify the given resource. Attribute **`identifier`** must be filled and identifies the resource to be modified. That operation is permited only when the user with given **`token`** is the resource owner. The **`resource`** should contain only attributes to be modified.

- **void deleteResource(SecurityToken token, String resourceIdentifier)**

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

- **Collection<ResourceSummary> listResources(SecurityToken token)**

  List of resource summaries managed by Shongo, that a user with given **token** is entitled to see.

- **Resource getResource(SecurityToken token, String resourceIdentifier)**

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

- **ResourceAllocation getResourceAllocation(SecurityToken token, String resourceIdentifier, Interval interval)**

  Get the allocation of resource object for specified **resourceIdentifier** in given **interval** that a user with given **token** is entitled to see. See **ResourceAllocation** for details.

4.5 Reservations

- **String createReservationRequest(SecurityToken token, AbstractReservationRequest reservationRequest)**

  Create a new reservation. The new reservation identifier is returned as a result. The **reservationRequest** must contain all attributes marked as **Required**.

- **String modifyReservationRequest(SecurityToken token, AbstractReservationRequest reservationRequest)**

  Modify the reservation. Attribute **identifier** must be filled and identifies the reservation request to be modified. The **reservationRequest** should contain only attributes to be modified.

- **String deleteReservationRequest(SecurityToken token, String reservationRequestIdentifier)**

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

- **Collection<ReservationRequestSummary> listReservationRequests(SecurityToken token)**

  List all the reservation requests that a user with given **token** is entitled to see. Only the lightweight definitions of reservation requests are returned, see **ReservationRequestSummary** for details.

- **AbstractReservationRequest getReservationRequest(SecurityToken token, String reservationRequestIdentifie**

  Get the reservation request for specified **reservationIdentifier** that a user with given **token** is entitled to see. The returned object contains requested time slots, requested compartments, 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**.

- **Collection<Reservation> listReservations(SecurityToken token, String reservationRequestIdentifier)**

  List allocated reservations for reservation request with given **reservationRequestIdentifier** that a user with given **token** is entitled to see.

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

  Get reservation with given **`reservationIdentifier`** that a user with given **`token`** is entitled to see.

4.6 Resource Control

TODO

Chapter 5

Connector API Specification

5.1 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 [4]. The communication is **synchronous**, i.e., the controller sends a command to a connector and waits until the connector replies. All messages are encoded using the FIPA SL content language [5]. 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.jade.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 5.5, might be specified by the following class:

```
package cz.cesnet.shongo.jade.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 [6]. 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| [6] 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 [5], 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.jade.ontology| interface and be tagged by the |Concept| interface.

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

5.2 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 3.4)
  - **`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:

  - `List<RoomUser> users` (Required)
    List of allowed users.
  - `boolean allowGuests` (Required)
    A flag indicating whether to allow guest users to join the room.
  - `int licenseCount` (Required)
    Number of licenses that multipoint server can utilize for this room.
  - `RoomLayout layout` (Required)
    The default room layout (used for all participants who did not specify a layout of their own choice).
  - `List<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.
  - `DateTime startTime` (**ReadOnly**)
    Date and time when the room was (or is to be) started.
  - `Reservation reservation` (**ReadOnly**)
    Reservation for which this room was created (to satisfy use-case 23).
  - `Technology type` (**ReadOnly**)
    Type of the room.

- `enum RoomLayout`

  Layout of a virtual room.

  Enumeration values:

  - `SingleParticipant` (only a single, fixed participant is displayed)
  - `VoiceSwitchedSingleParticipant` (only a single, currently speaking participant is displayed)
  - `SpeakerCorner` (a fixed participant is in the upper-left corner, other participants around)
  - `VoiceSwitchedSpeakerCorner` (the currently speaking participant is in the upper-left corner, other participants around)
  - `Grid` (all participants are spread in a regular grid)

- `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.
  - `CompressionAlgorithm compression` (**Optional**)
    Algorithm used to compress |data|.

- `class ContentType`

  Description of a media type. Any MIME Media Type listed by IANA [7], 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|).

- `enum CompressionAlgorithm`

  A compression algorithm used to compress data files.

  Enumeration values:

  - `ZIP` (zip compression, as specified by the application/zip MIME type)
  - `RAR` (rar archive)
  - `TAR_GZIP` (a gzip-compressed tar archive)
  - `TAR_BZIP2` (a bzip2-compressed tar archive)

5.3 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.

5.4 Multipoint Device

5.4.1 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 |room| parameter specifies the room settings, see the |Room| definition. Returns an identifier of the created room, unique within the device, to be used for further identification of the room as the |roomId| parameter.

- `modifyRoom(SecurityToken token, String roomId, Map attributes)`

  Modifies a room identified by |roomId|. The |attributes| map specifies |Room| attribute names mapped to new values.

- `deleteRoom(SecurityToken token, String roomId)`

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

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

  Gets current settings of a room exported to XML.
  TODO: Specify schema of the exported XML document in RelaxNG. It should contain at least room name, technology (H.323/SIP/Connect...) settings, and version of the format of the exported document (for further extensions).

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

  Sets up a room according to given |settings| previously exported by the |exportRoomSettings| method.

5.4.2 User Management

- **`List<RoomUser> listRoomUsers(SecurityToken token, String roomId)`**
- **`RoomUser getRoomUser(SecurityToken token, String roomId, String roomUserId)`**

  Gets user information and settings in a room.

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

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

- **`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.

5.4.3 Room Content Management

- **`MediaData getRoomContent(SecurityToken token, String roomId)`**

  Gets all room content (e.g., documents, notes, polls, etc.) as a single archive (see the |compression| attribute of the returned object).

- **`addRoomContent(SecurityToken token, String roomId, String name, MediaData data)`**

  Adds a data file to room content under a given name.

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

  Removes a file of a given name from room content.

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

  Clears all room content.

5.4.4 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. See the |contentType| of the returned object to get the image format returned.

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

  Gets a snapshot of the video stream that a user is sending in a room. See the |contentType| of the returned object to get the image format returned.

5.4.5 Recording

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

  Immediately starts recording in a room to format |format| using a given |layout| (or the default room layout, if |layout| is not specified). 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|.

- `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|.

- `notifyParticipants(SecurityToken token, int recordingId)`

  Sends an e-mail to all non-anonymous participants present in the room recorded. Participants present in any moment of the recording must be notified, not just the registered users.

- `downloadRecording(SecurityToken token, String downloadURL, String targetPath)`

  Starts downloading a recording from |downloadURL|. The recording is stored on the server under |targetPath|.

- `deleteRecording(SecurityToken token, int recordingId)`

  Deletes a given recording. The |recordingId| parameter is an identifier previously returned by |startRecording|. 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.

5.5 Endpoint Device

- **`dial(SecurityToken token, String server)`**
  Dials a server.
- **`resetDevice(SecurityToken token)`**
  Resets the device.

5.6 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 6

Deployment

This chapter describes the deployment of Shongo to a new domain.

6.1 Applications

Shongo consists of the following applications:

1. **Controller** represents a command-line application which should be launched in single instance for each domain and it acts as the main Shongo application for the domain.
2. **Connector** represents a command-line application which can be launched in multiple instances for each domain and each instance can manage one or multiple resources.
3. **Client** is command-line interface to a controller which can be used to setup resource database and to create reservation requests. Multiple instances of controller clients can run at the same time.

6.2 Installation

You must install Shongo to each machine where you want to launch a **controller**, **connector** or **client**. To install Shongo you need to get the source code. To get the Shongo source code you need to have Git[1] installed and use the following command:

```
git clone username@homeproj.cesnet.cz:shongo
```

To get an username and password ask at `martin.srom@cesnet.cz`.

6.2.1 Controller and connector

To build and launch **controller** or **connector** you need to have Java Platform (JDK)[2] and Maven[3] installed (preferred Maven version is 2.2.1). Enter the following directory:

```
cd <repository>/sw/shongo
```

[1] Git fast version control `http://git-scm.com/`
[2] Java Platform (JDK) `http://www.oracle.com/technetwork/java/`
[3] Apache Maven Project `http://maven.apache.org/download.html`

And type the following command:

```
mvn package
```

Controller and **connector** should be successfully built and tested.

6.2.2 Client

To launch **client** you need to have Perl[4] installed and also the following perl modules:

1. RPC::XML
2. XML::Twig
3. Text::Table
4. DateTime::Format::ISO8601
5. JSON

On Ubuntu/Debian system, Perl is installed by default and the modules can be installed by the following command:

```
sudo apt-get install librpc-xml-perl libxml-twig-perl \
        libtext-table-perl libdatetime-format-iso8601-perl libjson-perl
```

All applications (**controller**, **connector** or **client**) can be launched by entering the following directory:

```
cd <repository>/sw/shongo
```

And type the **`./<application>.sh`** command:

```
./controller.sh
./connector.sh
./client.sh
```

6.3 Controller

Controller by default runs on `localhost` interface (`127.0.0.1`) with XML-RPC server on port `8181` and Jade middle-ware on port `8282`. To change the default settings command-line attributes can be used:

```
./controller.sh --host <host> --jade-port <jade-port> --rpc-port <port>
```

Another way to change the default settings is to use a configuration file which has more options. Create file `<repository>/sw/shongo/controller.cfg.xml` which should contain:

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    ...
</configuration>
```

The following options are available in the configuration file:

- Configuration of controlled domain. It's code **`name`** and name of **`organization`** which run's the controller.

[4] Perl `http://www.perl.org/get.html`

```
<domain>
     <name>cz.cesnet</name>
     <organization>CESNET, z.s.p.o.</organization>
</domain>
```

- Configuration of XML-RPC (**host** and **port**).

```
<rpc>
    <host>127.0.0.1</host>
    <port>8181</port>
</rpc>
```

- Configuration of XML-RPC authentication and authorization. An **authorization-server** can be specified.

```
<security>
    <authorization-server>
        https://hroch.cesnet.cz/phpid-server/oic/
    </authorization-server>
</security>
```

- Configuration of Jade middle-ware (**host** and **port**).

```
<jade>
    <host>127.0.0.1</host>
    <port>8282</port>
</jade>
```

- Configuration of reservations. Maximum duration of resource reservation and alias reservation can be configured.

```
<reservation>
    <resource>
        <max-duration>P6D</max-duration>
    </resource>
    <alias>
        <max-duration>P1Y</max-duration>
    </alias>
</reservation>
```

- Configuration of controller's worker which periodically runs scheduler and preprocessor. A **period** in which the worker runs and the length of a working **interval** can be configured. The `PT10S` period means that every 10 seconds scheduler and preprocessor is executed. The `P31D` interval means that preprocessor and scheduler will process only reservation requests which are 31 days ahead.

```
<worker>
    <period>PT10S</period>
    <interval>P31D</interval>
</worker>
```

- Configuration of executor which periodically checks for allocated compartments and executes them. A **period** can be configured in which the executor performs checking and also the **lookup-ahead** which is used when searching for allocated compartments to be executed (the executor executes only compartments which take place in interval starting at current date/time and with length of **lookup-ahead**). Option **start** defines duration by which is modified the starting date/time of the executed compartment and the **end** modifies the ending date/time respectively (both PT-30S means that the compartment will be started and ended 30 seconds beforehand). Periods **waiting-start** and **waiting-end** specify how often the compartment executor checks whether the compartment should actually be started or stopped.

```
<executor>
    <period>PT15S</period>
    <lookup-ahead>PT3M</lookup-ahead>
    <compartment>
        <start>PT-30S</start>
        <end>PT-30S</end>
        <waiting-start>PT10S</waiting-start>
        <waiting-end>PT10S</waiting-end>
    </compartment>
</executor>
```

6.4 Connector

Connector by default connects to the controller on 127.0.0.1:8282 through Jade middle-ware and the connector runs Jade middle-ware on 127.0.0.1:8383. To change the default settings command-line attributes can be used:

```
./connector.sh --controller <host>:<port> --host <jade-host> --port <jade-port>
```

Another way to change the default settings is to use a configuration file which has more options. Create file <repository>/sw/shongo/connector.cfg.xml which should contain:

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    ...
</configuration>
```

The following options are available in the configuration file:

- Configuration of the connection to the controller (**host** and **port**). The connection to the controller is periodically checked and in case of failure the connection is re-established. The option **connection-check-period** specifies how often the check is performed.

```
<controller>
    <host>127.0.0.1</host>
    <port>8282</port>
    <connection-check-period>PT10S</connection-check-period>
</controller>
```

- Configuration of Jade middle-ware (**host** and **port**).

```
<jade>
    <host>127.0.0.1</host>
    <port>8383</port>
</jade>
```

- Configuration of connector agent instances which are automatically started when the connector is launched. Each connector can run multiple connector agents.

```
<instances>
    <instance>
        ...
    </instance>
    <instance>
        ...
    </instance>
    ...
</instances>
```

Example of connector agent configuration for Tandberg Codec C90:

```
<instance>
    <name>c90</name>
    <device>
        <connectorClass>cz.cesnet.shongo.connector.CodecC90Connector</connectorClass>
        <host>147.251.54.102</host>
        <port>22</port>
        <auth>
            <username>(username)</username>
            <password>(password)</password>
        </auth>
    </device>
</instance>
```

Example of connector agent configuration for Codian MCU 4515:

```
<instance>
    <name>mcu</name>
    <device>
        <connectorClass>cz.cesnet.shongo.connector.CiscoMCUConnector</connectorClass>
        <host>mcuc.cesnet.cz</host>
        <auth>
            <username>(username)</username>
            <password>(password)</password>
        </auth>
    </device>
</instance>
```

6.5 Client

Client can be started and connected to a controller by the following commands:

```
./client.sh --connect 127.0.0.1 [--testing-access-token]
./client.sh --connect 127.0.0.1:8181 [--testing-access-token]
```

Option `--testing-access-token` forces the client to skip the user authentication and to use testing access token which is automatically validated in the controller. The available commands in the client shell can be displayed by typing:

```
shongo> <TAB>
```

Or by typing:

```
shongo> help <ENTER>
```

Bibliography

[1] T. Berners-Lee, R. Fielding, L. Masinter, Rfc 3986, uniform resource identifier (uri): Generic syntax (2005).
URL http://rfc.net/rfc3986.html

[2] D. Winer, XML-RPC Specification (1999).
URL http://www.xmlrpc.com/spec

[3] OpenID Connect Basic Client Profile 1.0.
URL http://openid.net/specs/openid-connect-basic-1_0.html

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

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

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

[7] Internet Assigned Numbers Authority, MIME Media Types (March 2012).
URL http://www.iana.org/assignments/media-types/index.html

Appendix A

Client Usage

Client can be started and connected to a controller by the following commands:

```
./client.sh --connect localhost --testing-access-token
```

New resource can be interactively created by typing:

```
shongo> create-resource
```

New resource can be automatically created by typing:

```
shongo> create-resource -confirm { \
    class: 'DeviceResource', \
    name: 'mcu', \
    allocatable: 1, \
    technologies: ['H323'], \
    mode: { \
        connectorAgentName: 'mcu' \
    }, \
    capabilities: [{ \
        class: 'VirtualRoomsCapability', \
        portCount: 100 \
    }, { \
        class: 'AliasProviderCapability', \
        technology: 'H323', \
        type: 'E164', \
        pattern: '9500872[dd]', \
        restrictedToOwnerResource: 1 \
    }] \
}
```

List of existing resources can be showed by typing:

```
shongo> list-resources
```

Detail of existing resource can be showed by typing:

```
shongo> get-resource <resource-identifier>
```

Summary of allocation for existing resource can be showed by typing:

```
shongo> get-resource-allocation <resource-identifier> [-interval 2012-01-01/P1Y]
```

New reservation request can be interactively created by typing:

```
shongo> create-reservation-request
```

New reservation request can be automatically created by typing:

```
shongo> create-reservation-request -confirm { \
    class: 'PermanentReservationRequest', \
    name: 'Example', \
    resourceIdentifier: 'shongo:cz.cesnet:1', \
    slots: [{ \
        start: '2012-01-01T12:00', \
        duration: 'PT4M' \
    }] \
}
```

List of existing reservation requests can be showed by typing:

```
shongo> list-reservatin-requests
```

Detail of existing reservation request can be showed by typing:

```
shongo> get-reservation-request <reservation-request-identifier>
```

Allocated reservation(s) for existing reservation request can be showed by typing:

```
shongo> get-reservation-for-request <reservation-request-identifier>
```

Or by typing:

```
shongo> get-reservation <reservation-identifier>
```

For scripting purposes it is useful to run the client with command(s) which should be executed (the client will not run the shell in this case and exits immediately):

```
./client.sh --connect localhost --testing-access-token \
    --cmd "list-resources" \
    --cmd "list-reservation-requests"
```

Appendix B

Controller API Usage

B.1 Perl programming language

B.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");
}
```

B.1.2 Create reservation

```
$response = $client->send_request(
    'Reservation.createReservationRequest',
    RPC::XML::string->new('1e3f174ceaa8e515721b989b19f71727060d0839'), # access token
    RPC::XML::struct->new(
        'class' => RPC::XML::string->new('ReservationRequest'),
        'slot' => RPC::XML::string->new('20120101T12:00/PT2H'),
        'name' => RPC::XML::string->new('test'),
        'purpose' => RPC::XML::string->new('EDUCATION')
        ...
    )
);
```

Response

```
<struct>
  <member>
    <name>class</name>
    <value><string>ReservationRequest</string></value>
  </member>
  <member>
    <name>id</name>
    <value>
      <string>shongo:cz.cesnet:1</string>
    </value>
  </member>
  <member>
    <name>slot</name>
    <value><string>20120101T12:00:00/PT2H</string></value>
  </member>
  ...
</struct>
```

B.1.3 Modify reservation

```
$response = $client->send_request(
    'Reservation.modifyReservationRequest',
    RPC::XML::string->new('1e3f174ceaa8e515721b989b19f71727060d0839'),
    RPC::XML::string->new('shongo:cz.cesnet:1'),
    RPC::XML::struct->new(
        'description' => RPC::XML::struct->new() # set description to null
    )
);
```

Response

```
<struct>
  <member>
    <name>id</name>
    <value><string>shongo:cz.cesnet:1</string></value>
  </member>
  <member>
    <name>class</name>
    <value><string>ReservationRequest</string></value>
  </member>
  <member>
    <name>type</name>
    <value><string>20120101T12:00:00/PT2H</string></value>
  </member>
  ...
</struct>
```

B.1.4 List reservations

```
$response = $client->send_request(
    'Reservations.listReservationRequests',
    RPC::XML::string->new('1e3f174ceaa8e515721b989b19f71727060d0839')
);
```

Response

```
<array><data>
  <value><struct>
    <member>
      <name>id</name>
      <value><string>shongo:cz.cesnet:1</string></value>
    </member>
    <member>
      <name>class</name>
      <value><string>ReservationRequest</string></value>
    </member>
    <member>
      <name>type</name>
      <value><string>20120101T12:00:00/PT2H</string></value>
    </member>
    ...
  </struct></value>
</data></array>
```

B.1.5 Exception handling

Wrong class

```
$response = $client->send_request(
    'Reservations.listReservationRequests',
    RPC::XML::string->new('1e3f174ceaa8e515721b989b19f71727060d0839')
);
```

Response

```
<fault>
  <value><struct>
    <member>
      <name>faultString</name>
      <value><string>Class 'SecurityTokenX' is not defined.</string></value>
    </member>
    <member>
      <name>faultCode</name>
      <value><i4>10</i4></value>
    </member>
  </struct></value>
</fault>
```

Wrong attribute name

```
$response = $client->send_request(
    'Reservations.createReservationRequest',
    RPC::XML::string->new('1e3f174ceaa8e515721b989b19f71727060d0839'),
    RPC::XML::struct->new(
        'typeX' => RPC::XML::string->new('PERMANENT')
    )
);
```

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>12</i4></value>
    </member>
  </struct></value>
</fault>
```

Wrong attribute value

```
$response = $client->send_request(
    'Reservations.createReservationRequest',
    RPC::XML::string->new('1e3f174ceaa8e515721b989b19f71727060d0839'),
    RPC::XML::struct->new(
```

```
        'purpose' => RPC::XML::struct->new(
            'class' => RPC::XML::string->new('SecurityToken')
        )
    )
);
```

Response

```
<fault>
  <value><struct>
    <member>
      <name>faultString</name>
      <value><string>Attribute 'purpose' in class 'ReservationRequest' has type
          'ReservationRequestPurpose' but 'SecurityToken' was presented.</string></value>
    </member>
    <member>
      <name>faultCode</name>
      <value><i4>13</i4></value>
    </member>
  </struct></value>
</fault>
```

Wrong enum

```
$response = $client->send_request(
    'Reservations.createReservationRequest',
    RPC::XML::string->new('1e3f174ceaa8e515721b989b19f71727060d0839'),
    RPC::XML::struct->new(
        'purpose' => RPC::XML::string->new('SCIENCEX')
    )
);
```

Response

```
<fault>
  <value><struct>
    <member>
      <name>faultString</name>
      <value><string>Enum value 'SCIENCEX' is not defined in enum
          'ReservationRequestPurpose'.</string></value>
    </member>
    <member>
      <name>faultCode</name>
      <value><i4>20</i4></value>
    </member>
  </struct></value>
</fault>
```

Appendix C

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.jade.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 )
```