# ECSA — Elearning Community Service Architecture

| REVISION HISTORY | | | |
|---|---|---|---|
| NUMBER | DATE | DESCRIPTION | NAME |
| 6af189a | 2016-09-23 | Separate out campusconnect and installation stuff. | Heiko Bernloehr |
| 7b6a2e6 | 2014-08-30 | Added links to official cookie specification. | Heiko Bernloehr |
| 73f70f9 | 2014-08-08 | ecs_hash_url now marked as DEPRECATED. | Heiko Bernloehr |
| 0e2c56b | 2014-04-03 | Broken link repaired. | Heiko Bernloehr |
| 73f05e5 | 2014-02-18 | Unicorn application server. | Heiko Bernloehr |
| c576ee8 | 2014-02-18 | Some minor text changes. | Heiko Bernloehr |

# Contents

# Chapter 1

# Overview

An ECSA is a service architecture for elearning based webservices. It provides mechanisms for communication and authorization between elearning systems among each other and management systems. This is implemented via a MOM.

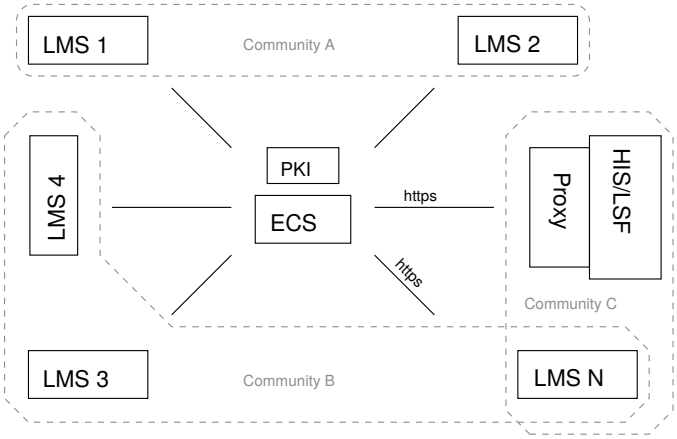



Figure 1.1: Components of an ECSA network.

An ECSA builds up of three primary components:

- The ECS (elearning community server) serves the core functionionality of an ECSA network. It provides named message resources to allow communication between all participants.
- An ECC (elearning community client) is a participant in an ECSA network. It has to be registered at ECS and must be able to talk to the ECS as a REST based client. This participant normally has a native implementation of the ECS interface. Our favourite ECCs are LMSs (learning management systems).
- An ECP (elearning community proxy) represents a special kind of participant. It serves as a proxy for a none ECSA compliant system so that such a system is able to participate in an ECSA network without ever knowing about it.

## 1.1 Sample usage scenario

Suppose you have several LMSs (learning management systems) and want to share courses between them. You decide not to interchange the real courses but only course links which consist of some meta data of the appropriate course especially a link formed by an URL pointing to the real course so you can call it through the WWW e.g.:

`http://ilias.freeit.de/goto.php?target=pg_26_43&client_id=ecs2`

Now it's possible for each LMS to communicate the released courses by the resources provided from the ECS to an explicit LMS (point to point) or to a community of LMSs (point to multipoint).

Because of the uniform application interface — there are only GET, PUT, DELETE and POST operations — receiving participants can fetch messages through a GET on the resource URL or sending messages by a POST on the resource URL (with some additional query parameters or header variables to point to the appropriate receivers).

To illustrate this we use the simple ECC application `curl` to send a message from one participant to another:

```
curl -i -H 'X-EcsAuthId: pid01' \
        -H 'X-EcsReceiverMemberships: mid02' \
        -H 'Content-Type: application/json' \
        -X POST \
        -d '{
              "name": "Mathematics II",
              "url" : "http://ilias...?target=pg_26_43&client_id=ecs2",
              ...
            }' \
        http://ecs.freeit.de/campusconnect/courselinks
```

In order to receive a message (in fifo mode) the receiving participant may call:

```
curl -i -H 'X-EcsAuthId: pid02' \
        -H 'Accept: text/plain; application/json' \
        -X GET \
        http://ecs.freeit.de/campusconnect/courselinks/fifo
```

Of course, there are several ways to operate on a resource.

# Chapter 2

# Participants

A particpant represents a legal client in an ECSA network.

## 2.1 Basic functionalities and requirements

### 2.1.1 Technology / Architecture

- has to communicate with the ECS as a REST client.
- HTTP 1.1 as transport and application protocol
- provide persistent connection (keep-alive)
- provide SSL/TLS transport layer
- has to use UTF-8

### 2.1.2 Authentication

- HTTP Basic auth
- X.509 Certificates (SSL/TLS client authentication)

## 2.2 Authorization

A client should be able to use a simple "one touch token" authorization through the ECS `sys/auths` resource. This token could be used to accomplish deligated authorization for accessing resources on participants of a common ECSA network. E.g. in redirecting users clicking on course links or in direct communicating of participants.

## 2.3 Ressource extensions / alterations

To make resource extensions and alteration possible the clients have to easily permit

- additional ressources
- extensible data formats
- Postels's Law (robustness principle): *Be conservative in what you send; be liberal in what you accept.*

- versioning through request and response header (content negotiation)
  - Accept: application/vnd.my-format.v1+json
  - Accept: application/vnd.my-format.v2+json

## 2.4 Web interfaces

- Interface for ECS configuration data

## 2.5 Communication procedures / scenarios

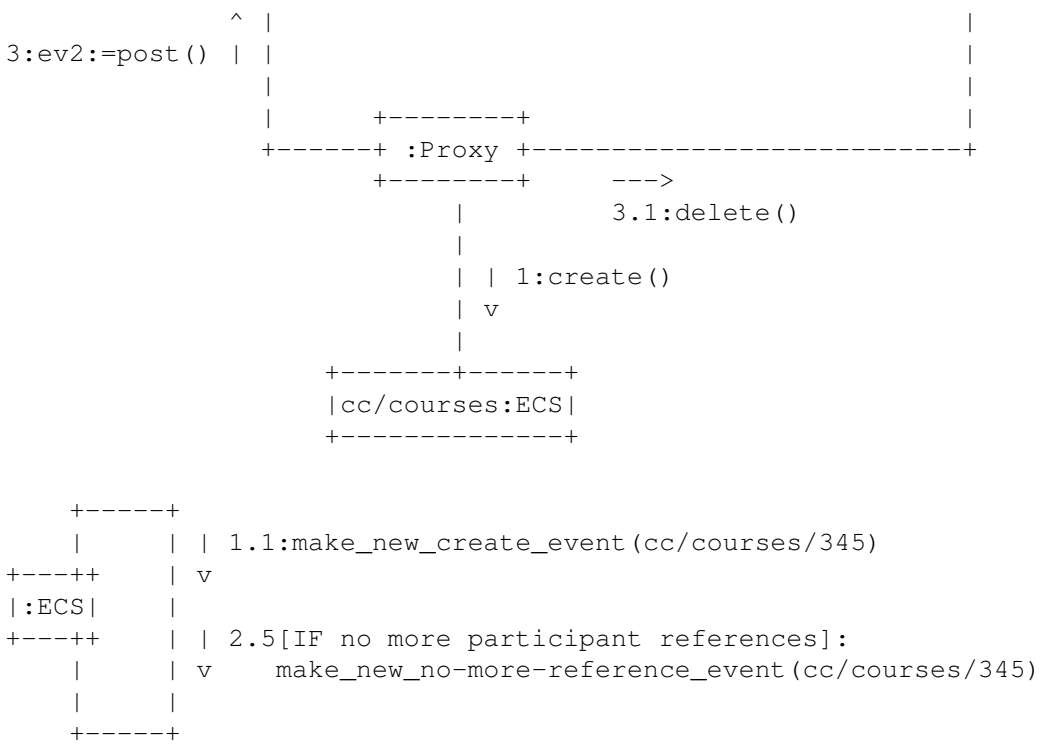
In order to take part in an ECSA network a participant has to communicate with the ECS and other participants in different ways.

### 2.5.1 Direct participant to participant communication

This communication procedure takes place if a participant wants to get resource data directly from another participant. Normally all participants communicate only with the ECS and not face to face to each other. For example this could be necessary if you're sharing a central ECS with other organizations and you are not allowed to give away sensible data out of the control of your organization. Of course therefore the communicating participants have to be in the controlling area of your organization.

Assuming following situation: A Proxy (first participant) wants to send course data to a LMS (second participant). This message holds sensible data, which doesn't have to leave the controlling area of the organization by law. Both participants were controlled by the organization.

This procedure guarantiee that the appropriate course data will remain on the proxy until the LMS has successfully fetched the data.




**1**

The proxy creates a new course representation on the ECS, which was addressed to the LMS. The proxy doesn't store the sensible data there, instead it stores an URL were the real data could be fetched.

**1.1**

The ECS makes a new create event on the event queue of the receiving LMS, storing the new generated resource URL `cc/courses/345` in it.

**2**

Then the LMS fetches (POST) this event message from its event resource (`sys/events/fifo`) of ECS , which gives it a new or updated coursedata URL on ECS. This would be `cc/courses/345` with `Content-Type: text/uri-list` (mime type see rfc2483).

**2.1**

Now the LMS takes this URL and fetches (GET) it from ECS (the LMS only fetches the message via a GET, so that the message will still be there). Only now the LMS gets the real resource URL to fetch the desired course data from the proxy. This url maybe an obscured url like `https://.../746354389534`

**2.2**

Next the LMS fetches (POST) a one touch token from the `sys/auths` resource of ECS in case the proxy use it for authorization against ECS.

**2.3**

Then the LMS gets (GET) the actual course data from the proxy URL provided by the received message in 2.1 .

**2.3.1**

The proxy returns the coursedata to the LMS if the auth token (auth.hash) provided in 2.3 is still available at `sys/auths` resource on ECS.

**2.4**

When it will get back the course resource representation in 2.3 successfully, it deletes (DELETE) the message `cc/courses/345` received in 2.1 on ECS.

**2.5**

If there are no further references on `cc/courses/345` ECS makes a new no-more-reference event for `cc/courses/345` addressed to the proxy (original sender).

**3**

Proxy is fetching (POST) its events queue.

**3.1**

If the received event in 3 was a no-more-reference event for `cc/courses/345`, Proxy knows, that nobody further references `cc/courses/345`. This tells the Proxy, that every addressed participants have fetched `cc/courses/345` on ECS and hence all participants has fetched `https://.../746354389534`. If this resource was not configured persistent Proxy could delete `https://.../746354389534`.

# Chapter 3

# ECS

The elearning community server (ECS) is designed as a message oriented middleware (MOM) and is implemented as a REST conform application.

Because the ECS was born in an elearning context the following definition shows consideration of that. Nevertheless the ECS could be used in other areas of responsibility.

The ECS groups its participants in so called *communities*. Participants could address each other only if they share a community. Therefore they could address an explicit participant, a participant list or the whole members of the community (see ECS API for details).

All participants have to register at the ECS. Every registered participant has access to at least three system resources (`/sys/memberships`, `/sys/events`, `/sys/auths`) to get informed and take part at a ECSA network. To design/map your specific application communication you can create as many application resources you want.

## 3.1 HTTP Header

### 3.1.1 ECS specific headers

**X-EcsAuthId**
Has to be a valid participant id. In a standard ECS configuration this HTTP header will be attached by the authentication process running on the proxy server.

**X-EcsReceiverCommunities**
Has to be a valid community id/ids or community name/names. Adresses all participants joined the comimunity/communities. You are able to note multiple communities, either by name or by id, spaced by comma. Only allowed by POST.

**X-EcsReceiverMemberships**
Has to be a valid membership id/ids. Adresses all listed memberships. You are able to note multiple memberships spaced by comma. Only allowed by POST.

**X-EcsSender**
Describes the sender of a message. If you GET a resource this header variable shows the sender membership id. Additionally the ECS sets the X-EcsReceiverCommunities variable to the community from which you have received the message. If the message reach you from several communities X-EcsSender show you a comma separated list of membership ids representing the appropriate membership id of the sender in these communities. In this case the X-EcsReceiverCommunities variable would also represent a comma seperated list of a corresponding community ids.

**X-EcsQueryStrings**
Used to provide querystrings.

### 3.1.2 HTTP standard header

**Accept**
Content-Types that are acceptable.

**Content-Type**
The mime type of the body of the request (used with POST and PUT requests).

**If-None-Match**
Allows a 304 Not Modified to be returned if content is unchanged.

**Cookie**
An HTTP cookie previously sent by the server with Set-Cookie (Wikipedia, RFC6265).

**Content-Type**
The mime type of this content.

**ETag**
An identifier for a specific version of a resource.

**Location**
Used in redirection, or when a new resource has been created.

**Set-Cookie**
An HTTP cookie (Wikipedia, RFC6265).

## 3.2 HTTP return codes

**200**
Successful GET.

**201**
Successful POST.

**304**
A *Not Modified* response on a conditional GET. This means the requested resource has not been changed.

**404**
Resource not available.

**4xx**
General client side error.

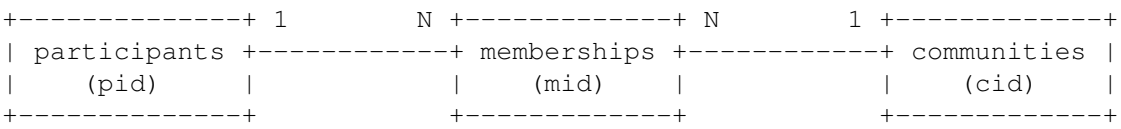
**5xx**
General server side errors.

## 3.3 Addressing

In order to communicate to each other you have to provide a unique address. These addresses can either be a so called membership id or a community id or community name.

### 3.3.1 Membership IDs

These are unique ids in the scope of an ECS. They establish a relationship between a participant and a community:




Therefore a participant can be associated to different communities. Every participant can inquire his membership ids by calling the memberships resource.

### 3.3.2 Community names and ids

A community can be referenced by his community id (cid) or his community name. If you address a community you implicit address all members of the community. This applies also to the sender joining the receiver community if the sender has set his community_selfrouting flag (default off), otherwise the sender will be implicitly excluded from the receiver list. Every participant can inquire his communities memberships by calling the memberships resource.

### 3.3.3 Create a resource

If you want to POST to a resource you have to provide either a `X-EcsReceiverMemberships` or `X-EcsReceiverCommunities` header or both together.

If you want to address a single membership or a dedicated number of memberships you have to set the `X-EcsReceiverMemberships` header. This header can have a list of values, e.g.

```
X-EcsReceiverMemberships: 3,6,47
```

If you want to address a community you have to set the `X-EcsReceiverCommunities` header. This header can have a list of values, e.g.

```
X-EcsReceiverCommunities: SWS,23,25
```

### 3.3.4 Get a resource

If you GET a resource then the ECS set the `X-EcsSender` and the `X-EcsReceiverCommunities` header to show you from whom and where your received message comes. If there is a list of `X-EcsReceiverCommunities` values than there is also a list of corresponding `X-EcsSender` values, i.e. the sending participant is member of multiple communities and addressed his message to multiple communities also, e.g.

```
X-EcsSender: 3,19
X-EcsReceiverCommunities: UnisBW,SUV
```

This means that this message is addressed to you through two communities (UnisBW, SUV) and the sender has the membership id 3 in UnisBW and 19 in SUV.

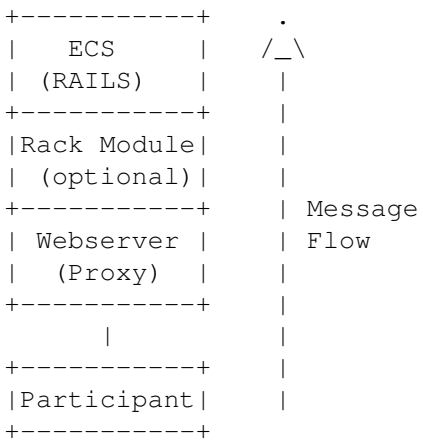
## 3.4 Community selfrouting

If community selfrouting is activated at the participant (administration area) you can decide if you also want to receive the message which you send to an appropriate community, i.e. you get an event notification (if events on this resource is activated) and you get it listed by its list resource and could access it through its queue resource. Of course, as sender of the message you can always access it by its message resource.

## 3.5 Authentication

All participants have to be authenticated in order to use ECS services. A participant is deemed to be authenticated if the `X-EcsAuthId` header is set and the ECS knows it. The real authentication take place in front of the ECS, normally at the Webserver. But this depends on configuration/installation of ECS:

**Message flow through ECS application.**




Currently supported authentication methods:

- Basic Auth
- X.509 certificates

## 3.6 Anonymous participants

The creation of a new anonymous participant automatically takes place by every call to an ECS resource if the calling participant didn't set `X-EcsAuthId` or `Cookie` header, by setting a `Set-Cookie` header in the response. On subsequent calls the participant has to provide this cookie in a `Cookie` header in order to be identified as the previously calling participant. Additionally those participants were automatically joined to the *public* community. Further their lifetime will be limited and all resources will be silently deleted after this lifetime becomes zero. With succesional accesses to ECS this lifetime will be refreshed. For general cookie handling see Wikipedia and RFC6265. See also ECS curl examples.

## 3.7 System resources

### 3.7.1 Events

Provides a general queue which accumulates the resource tasks: creation, deletion and renewal. Available representations are application/json and application/xml. It's recommended to use the events queue to supervise all your possible application specific resources. Further you only have to poll the events queue in order to supervise all your application specific resources and this further take down system load.

Remark: If you wisely decide to use the events queue to supervise your application specific resources you have to manage the validity of events queue yourself, i.e. you shouldn't additionally poll your application specific resources directly, because then you will get stale events in the events queue.

**/sys/events**
: GET provides a list of events for the appropriate calling participant. Optionally the query string parameter *count* could be used to limit the amount of returned events.

**/sys/events/fifo**
GET provides an event (the oldest one) for the appropriate calling participant. Optionally the query string parameter *count* could be used to extend the amount of returned events. POST provides an event (the oldest one) for the appropriate calling participant and removes it from the events queue. Optionally the query string parameter *count* could be used to extend the amount of returned events.

Following a sample representation in JSON:

```
[
  {
    "status": "created",
    "ressource": "numlab/exercises/7"
  },
  {
    "status": "destroyed",
    "ressource": "numlab/exercises/3"
  }
]
```

### 3.7.2 Memberships

Provides information of the affiliation of the calling participant to the available communities. Available representations are `application/json` and `application/xml`.

**/sys/memberships**
GET provides a list of memberships for the appropriate calling participant. It implies all participants joining an appropriate community including the caller itself.

With the `itsyou` key the caller of the `/sys/memberships` resource will be informed which participant in the different communities is assigned to him.

Following a sample representation in JSON:

```
[
  {
    "community": {
      "name": "cc_courselinks",
      "description": "CampusConnect courselinks."
    },
    "participants": [
      {
        "name": "ILIAS-ECS Client 1",
        "itsyou": true,
        "org": {
          "name": "Leifos",
          "abbr": "LEI"
        },
        "mid": 1,
        "pid": 1,
        "description": "Development participant.",
        "dns": "n/a",
        "email": "meyer@leifos.com"
      },
      {
        "name": "FreeIT.de Testparticipant",
        "itsyou": false,
        "org": {
          "name": "FreeIT Softwaredevelopment.",
          "abbr": "FreeIT"
```

```
        },
        "mid": 2,
        "pid": 4,
        "description": "A general test participant.",
        "dns": "n/a",
        "email": "Heiko.Bernloehr@FreeIT.de"
      },
      {
        "name": "ILIAS-ECS Client 2",
        "itsyou": false,
        "org": {
          "name": "Leifos",
          "abbr": "LEI"
        },
        "mid": 3,
        "pid": 7,
        "description": "",
        "dns": "n/a",
        "email": "meyer@leifos.com"
      }
    ]
  }
]
```

### 3.7.3 Auths

This means authorization through one touch tokens. Provides a mechanism to grant each participant authorization to consume services from any service-providing-participant in an ECS network.

The interface is the same as for application specific resources. If you want to create an authorization token, you have to provide at least a `realm` (authorization context) or a `url` (authorization context, DEPRECATED):

```
curl ... -X POST -d '{"realm":"authorization context string"}' https://.../sys/auths
```

and you will get back something like this:

```
{
  "hash": "5a944e72346e6e3102d32ccfecc18862d23e1dc0",
  "sov": "2011-03-08T23:25:27+01:00",
  "eov": "2011-03-08T23:26:27+01:00",
  "url": "authorization context string",
  "realm": "authorization context string",
  "abbr": "LEI",
  "pid": 35
}
```

**hash**
: provides the authorization token (one touch token)

**sov**
: stands for start of validation

**eov**
: stands for end of validation

**url**
: provides the authorization context (DEPRECATED)

**realm**
: provides the authorization context (replaces url)

**abbr**
: provides an abbreviation of the participant which has been created the authorization token (DEPRECATED, use pid as reference key in `sys/memberships` representation to get participant information)

**pid**
: provides the participant id of the participant which has been created the authorization token

You're allowed to set the `sov` and/or `eov` to determine the validity period of the authorization token. If you do not, the validity period is set to one minute starting at current time.

The recommended way to fetch an authorization token when knowing the one touch hash:

```
curl .... -X DELETE https://.../sys/auths/<one touch hash>
```

This will return the auths representation (same structure/form as when creating; see above) and delete it server side. If the authorization token is outimed, i.e. the current time is not between `sov` and `eov`, you will get back a return code 409 (conflict) and following descriptional text in the body: *Authorization token outtimed*.

## 3.8 Application specific resources

All application specific resources have to be configured at ECS. There are three types of application specific resources:

1. messages
2. lists
3. queues

Generally resources are an abtract concept:

- clearly identifiable (in an HTTP context through URLs)
- have one ore more representations (e.g. JSON, XML, text, . . . )

According to resources it plays no role how a representation is produced. It could be done by returning a static file or running a complex server side application, that doesn't matter. Furthermore by looking at a resource you can't conclude how the representation has been made. An evaluation of a resource based on internal operations and circumstances, it is thus also negligible, and even be inadmissible.

### 3.8.1 Resource structure

```
/<projectnamespace>/<name>
/<projectnamespace>/<name>/details
/<projectnamespace>/<name>/<id>
/<projectnamespace>/<name>/<id>/receivers
/<projectnamespace>/<name>/<id>/details
/<projectnamespace>/<name>/fifo
/<projectnamespace>/<name>/lifo
```

### 3.8.2 Message resource

A message resource receives/saves messages for each participant. The participant can fetch (GET) his messages from the resource. A message resource could hold its messages enduringly (see Section 3.8.5), so new participants joining a community after a message has been sent to this community will also receive it.

**GET**
Returns message with status code 200.

**DELETE**
Deletes message and returns deleted resource representation with ststus code 200.

**PUT**
Renew message and returns with status code 200.

**POST**
Illegal call. Returns with status code 405 (Method Not Allowed).

Resource structure: `/<projectnamespace>/<name>/<id>`

#### 3.8.2.1 Subresource details

You can ask for detailed (meta) information of a posted message. Only the original sender or a receiver can do that:

**GET**
Returns details about the requested message.

Resource structure: `/<projectnamespace>/<name>/<id>/details`

You will get back something like this:

```
{
  "receivers": [
    {
      "itsyou": false,
      "mid": 1,
      "cid": 2
      "pid": 19,
    },
    {
      "itsyou": false,
      "mid": 4,
      "cid": 3
      "pid": 29,
    }
  ],
  "senders": [
    {
      "mid": 5
    },
    {
      "mid": 7
    }
  ],
  "url": "courselinks/10",
  "content_type": "application/json"
  "owner": {
    "pid": 3,
```

```
    "itsyou": true
  }
}
```

The "receivers" and "senders" have corresponding arrays: The first array entry in "senders" has been addressed the first array entry of "receivers" and so on.

### 3.8.3 List resource

**GET**
Returns URI message list with status code 200. If there are no messages to list the HTTP body will be empty (`Content-Length: 0`). The `Content-Type` will be `text/uri-list`. The URI list will be represented by relative references. URIs are specified in RFC3986.

**DELETE**
Illegal call. Returns with status code 405 (Method Not Allowed).

**PUT**
Illegal call. Returns with status code 405 (Method Not Allowed).

**POST**
Creates new message, returns with status code 201 and a HTTP header `Location:` providing the new message URI.

Resource structure: `/<projectnamespace>/<name>`

#### 3.8.3.1 Subresource details

Now it's possible to ask for detailed (meta) information of a list resource. All querystrings supported my normal list resources could be used. Only the original sender can do that:

**GET**
Returns details about all resource URIs listed.

Resource structure: `/<projectnamespace>/<name>/details`

You will get back something like this:

```
[
  {
    "senders": [ ],
    "receivers": [ ],
    "url": "courselinks/35",
    "content_type": "text/plain",
    "owner": {
      "pid": 3,
      "itsyou": true
    }
  },
  {
    "senders": [
      {
        "mid": 2
      }
    ],
    "receivers": [
      {
```

```
        "mid": 19,
        "cid": 2,
        "pid": 19,
        "itsyou": false
      }
    ],
    "url": "courselinks/36",
    "content_type": "text/plain",
    "owner": {
      "pid": 3,
      "itsyou": true
    }
  },
  {
    "senders": [
      {
        "mid": 2
      }
    ],
    "receivers": [
      {
        "mid": 19,
        "cid": 2,
        "pid": 19,
        "itsyou": false
      }
    ],
    "url": "courselinks/37",
    "content_type": "text/plain",
    "owner": {
      "pid": 3,
      "itsyou": true
    }
  }
]
```


The first element of the returned array of the details list subresource probably needs some explanation. Both senders and receivers are empty lists. This means that the appropriate message isn't any more addressed to any participant. This further implies that all participants which had been addressed in the past have been received the message from their appropriate resource. But why was the message then not deleted ? Because the resource has been configured to be "postrouted". If that has not been the case, ECS would has been removed the message.

#### 3.8.3.2 Querystrings

To affect the returned representation you could assign the following querystrings to `X-EcsQueryStrings` header variable:

**receiver**
: It's possible to filter the returned index from a list resource to only those items to which the calling participant was formerly an addressed receiver (this is also the default, therefore it could be omited):

```
curl .... -H 'X-EcsQueryStrings: receiver=true' -X GET https://server/< ↩
    namespace>/<name>
```

**sender**
: It's possible to filter the returned index from a list resource to only those items to which the calling participant is the original sender:

```
curl .... -H 'X-EcsQueryStrings: sender=true' -X GET https://server/<namespace >/<name>
```

**all**
It's possible to filter the returned index from a list resource to show all messages either as addressed receiver or as original sender:

```
curl .... -H 'X-EcsQueryStrings: all=true' -X GET https://server/<namespace>/<name>
```

Using the `X-EcsQueryStrings` header variable is the recommended way to use querystrings. If you have to assign multiple querystrings please delimit the querystrings by comma (,).

Of course you can also specify the querystring by appending it to the end of the resource url, e.g.

```
curl .... -X GET https://server/<namespace>/<name>?all=true
```

### 3.8.4 Queue resource

The queue resource is modelled as a subresource of a list resource and it can operate either in lifo (last in first out) or fifo (first in first out) mode.

**GET**
Returns last (lifo) or first (fifo) message with status code 200. If there are no more messages in queue you will get an empty message (`Content-Length: 0`) and also status code 200.

**DELETE**
Illegal call. Returns with status code 405 (Method Not Allowed).

**POST**
Returns last (lifo) or first (fifo) message with status code 200 and deletes it. If there are no more messages in queue you will get an empty message (`Content-Length: 0`) and also status code 200.

**PUT**
Illegal call. Returns with status code 405 (Method Not Allowed).

Resource structure: `/<projectnamespace>/<name>/fifo` or `/<projectnamespace>/<name>/lifo`

### 3.8.5 Postrouting

If a resource has set its postroute flag, then all new participants will get postrouted this resource e.g. if you have posted some messages to a community named testcommunity and later joins a new participant to this community, he will get postrouted the former posted messages.

## 3.9 JSON-Schemas

A json media type for describing the structure and meaning of json documents. It's defined as an Internet-Draft working document of the IETF. There is also a homepage where you can start to discover more over JSON-Schemas.

All resource representations must have a `Content-Type` header variable containing an optional parameter `profile` pointing to its describing schema. For a respond on a `/campusconnect/courses` request this could be:

```
Content-Type: application/json; \
              profile=http://repo.or.cz/w/ecs.git/blob_plain/ \
              e5cc81b2201ac24294d2ac3e732f9ddac954cc84:/ \
              campusconnect/schemas/cc_courses.schema.json
```

It's up to you to validate and check the received data against the provided schema or to decide if you are able to process the format just receiving. There is always a version id inbetween the `profile` URL representing the commit id of the git repository. For the last `Content-Type` example this was `e5cc81b2201ac24294d2ac3e732f9ddac954cc84`. You can always ask for the latest schema of an appropriate resourse by using `HEAD` as the version id.

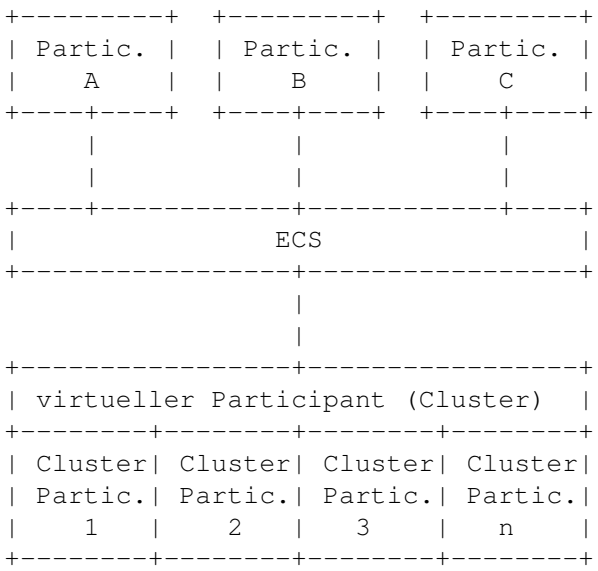
Of course you can use the schema of an appropriate resource for discovering the names and types of the data elements in order to match them dynamically to other internal meta data of your application.

## 3.10 Participant Cluster

The ECS is able to cluster participants. In the ECS network a cluster is seen as an ordinary participant.

### 3.10.1 Cluster building

First lets show the topology of a clustered ECS network:




1. The ECS registers a virtual participant. All cluster participants use this registration, i.e. the ECS doesn't know which cluster participant is communicating. This way you can scale your cluster easily by attaching another cluster participant also using the previously generated virtual participant registration. You don't have to make any further settings at ECS.
2. If you want to send a message to the cluster you only have to send it to the virtual participant. When all cluster participants compete against each other to get a message this would maybe the simplest resource access mode (message dispatching). Every cluster participant have to access the appropriate resource as a queue resource via DELETE method. This assures that every message could only be fetched by one cluster participant.

### 3.10.2 Cluster broadcasting

In order to explicitely communicate with a cluster participant we have to use a broadcasting mechanism. Every resource could be used as a broadcasting resource. It only depends on how the cluster participants access this resource. They have to do it like this:

1. Every cluster participant checks the broadcast resource as a queue resource with the idempotent GET method and decides by looking inside the message if this message is targeted to him. If it does belong to him he should compute and DELETE the message.
2. The ECS garbage collects the broadcast resource at a default time period.

# Chapter 4

# Filter plugins

Messages could be changed at runtime by so called filter plugins. These filters could be attached to 5 different queues and triggered by one of the actions hereafter. The filter queues were mapped to a special path under the filesystem:

1. Show filter. Triggered when calling a message/queue resource with GET. Filter path:
   `filter/<project-name-space>/<resource-name>/show/filter-name>`
2. Index filter. Triggered when calling a list resource with GET. Filter path:
   `filter/<project-name-space>/<resource-name>/index/filter-name>`
3. Create filter. Triggered when calling a list resource with POST. Filter path:
   `filter/<project-name-space>/<resource-name>/create/filter-name>`
4. Update filter. Triggered when calling a message/queue resource with PUT. Filter path:
   `filter/<project-name-space>/<resource-name>/update/filter-name>`
5. Delete filter. Triggered when calling a message/queue resource with DELETE. Filter path:
   `filter/<project-name-space>/<resource-name>/delete/filter-name>`

You're able to create as many filters you want. They will be all queued/concatenated in lexical order:

```
unfiltered +-------+   +-------+   +-------+ filtered
---------->| 1-fil |-->| 2-fil |...| n-fil |--------->
message    +-------+   +-------+   +-------+ message
```

If a filter was created and copied into the appropriate filesystem path, it would be automatically activated at runtime without additional configuration.

If there are any exceptions while reading (class loading) the filter the appropriate filter will be canceled and the processed message will be queued to the next one. If there are any exceptions while running the filter the appropriate filter will also be canceled and the processed message will also be queued to the next filter, but keep in mind, that all changes to the message prior to the occured exception will remain. You are always working with the original message (no copy). There will be error messages in the logfile of the form: "ERROR Filter Error: ...".

## 4.1 Template

In the <filter-name> directory must be at least a file called "filter.rb" with following structure:

```
module Filter
  def self.start
    ...
  end
end
```

ECS will call "Filter.start". From there its on you :) ECS will also load any file with ".rb" extention under the directory <filter-name> into the namespace of "Filter".

The ECS core provides the constant `FILTER_API` as an API for accessing ECS messages:

**FILTER_API.params**
It's a hash to access the qureystrings of message call:

```
http://ecs.rus.uni-stuttgart.de/numlab/exercises/23?properties=name, ↩
    description
...
elements = FILTER_API.params["elements"].split(",")
...
```

**FILTER_API.record**
This object provides access to the message body:

```
message = FILTER_API.record.body
```

# Chapter 5

# Interconnectivity

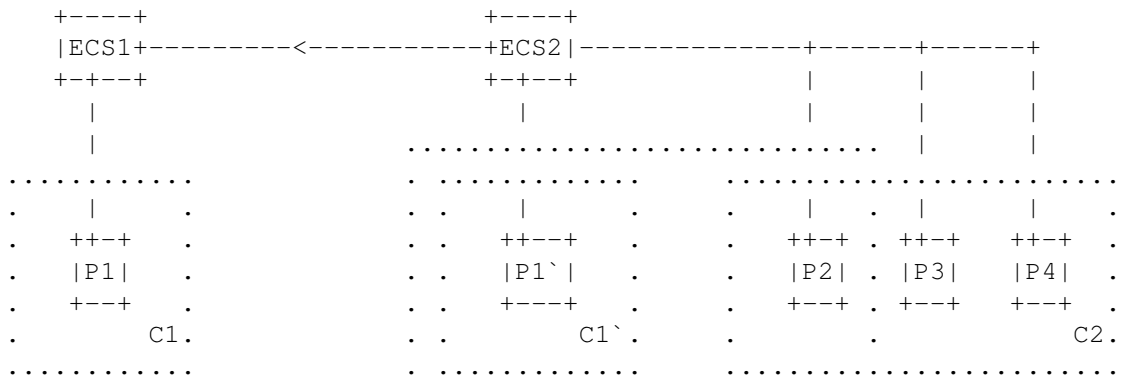
We want to interconnect several ECSs without any change to their joined participants. In other words this interconnection should be fully transparent to the appropriate participants. So there is no need for extra registration overhead to other ECSs. From the viewpoint of a participant there are just some more participants joining the communities of its primary ECS.

## 5.1 Necessary extensions to ECS

- In order to make use of event generation system the `sys/memberships` resource should be mapped to a normal application resource.
- Make ressources model an application resource `sys/resources`.
- participants should get a new attribute `foreign` which marks them as members from another ECS.
- resources should get a new attribute `foreign` which marks them coming from another ECS.
- communities should get a new attribute `foreign` which marks them coming from another ECS.
- ECS must provide full participant functionality.

## 5.2 Interconnection procedure

- To bidirectional interconnect two ECSs they both have to be participants of each other i.e. they have to register themself to each other.
- To unidirectional interconnect two ECSs only one of the ECS has to be registered at the other ECS. Then the registered ECS(2) provides its participants addressability to the participants of the registering ECS(1).




- As each normal participant, the ECS has to join the communities in interest. In opposite to a normal participant the ECS could not be seen by the other participants joining the same community hence they are not able to address an ECS directly but the ECS can determine them by reading its `sys/memberships` resource. Now the ECS is able to create the necessary remote participants and communities on his side and also join the new created remote participants to the right communities. This repeats whenever the ECS reads its `sys/memberships` resource. These remotly imported participants, communities and resources could not be altered in any way by the importing ECS.
- By calling `sys/resources` the ECS is also able to import and create all the resources from the remote ECS.
- Remotly added participants, communities and resources must not be imported by further ECSs. This prevents loops and retains full access determination of participants, communities and resources of the origin/owning ECS.

# Chapter 6

# Glossary

**DRY**
: don’t repeat yourself

**CRUD**
: create, read, update and delete

**ECC**
: elearning community client

**ECP**
: elearning community proxy

**ECS**
: elearning community server

**ECSA**
: elearning community service architecture

**IETF**
: internet engeneering task force

**LMS**
: learning management system

**MOM**
: message orientated middleware

# Chapter 7

# Index