Network Working Group
Internet-Draft
Expires: September 26, 2015

F. Maurer
March 25, 2015

The Federated Object Sharing Protocol
fosp-00

Abstract

This document describes the Federated Object Sharing Protocol (FOSP). FOSP is a protocol that allows sharing arbitrary data, setting access rights on the shared data and receiving notifications on changes. Furthermore, all features of FOSP can be used in a federated network, e.g. between multiple independned providers. It was designed to be the most simple solution that combines all these features.

Table of Contents

1. Introduction

1.1. Purpose

The Federated Object Storage Protocol (FOSP) is an application-level protocol for exchanging structured and unstructured data generated by users of different providers. It is designed to fulfill the needs of online social networks by combining necessary features into one simple protocol.

FOSP aims to provide three core features:

1. Store data online and support standard operations on it.

2. Enforce access control on the data, for users from different hosts, without central authentication.

3. Notify users when data is added, removed or has changed.

Furthermore, some non-functional requirements are imposed. The protocol needs to be data agnostic and should support structured, unstructured and metadata. It also must be as simple as possible, e.g. it must be easy to implement and it should be possible to write easy to deploy applications for it.

1.2. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [refs.rfc2119].

1.3. Requirements

FOSP builds on existing protocols and data formats. The transport protocol for FOSP messages currently is the WebSocket protocol and a HTTP binding is in the works. Other options may be added in the future. Objects are serialized into the JavaScript Object Notation (JSON). Besides these technical dependencies, FOSP is inspired by WebDAV and XMPP and has similarities to LDAP.

1.4. Terminology

This document uses a number of terminologies to refer to concepts found in FOSP.

provider An entity that provides storage space on the Internet for the data of users. It is identified by a fully qualified domain name.

server A FOSP server stores the data of users of a certain provider. The term server applies to the the software that processes request as well as to the machine that runs the software. For fault tolerance and load balancing purposes, a provider might deploy multiple servers.

client A FOSP client is a program a user uses to communicate with a FOSP server. It facilitates accessing the data of the user that is stored on the server.

message A message is the basic unit of communication in FOSP. Messages come in three different flavours.

request A request is a message sent from a client to a server. It is used to retrieve or alter data.

response A response is a message sent from a server to a client. It is always sent as an answer to a request and contains the status of the request and possibly data.

notification A notification is a message that is sent by servers when changes happen to an object.

object An object is the basic unit of data in FOSP. It consists of structured data expressed in JSON.

attachment An attachment is a binary or text file that is associated with an object. Each object can only have one attachment.

tree The objects of a users form a tree structure. Each object has exactly one parent object, except for the root object.

2. Overview

This section provides a short overview. It is intended to familiarize with the concepts of FOSP. It does not contain additional information that is not covered in the rest of the document and can be skipped.

2.1. User and providers

In the FOSP world, data is created by users. Users store their content on the servers of providers and can share it by setting the appropriate access rights for other users. Each user is registered with one provider. The user has an unique name at the provider that together with the domain name of the provider forms his or her identifier, similar to an Email address. However users can still share their content with friends that are registered with different providers.

Providers are companies or individuals that provide storage for users. A provider is identified by a fully qualified domain name. Similar to Email and XMPP, everybody can be a provider by simply hosting his or her own FOSP server. The servers of different providers communicate to allow users of different providers to share data.

2.2. Trees, Objects and Attachments

The basic data units that can be stored on the server have a well defined structure and we call them objects. We chose this name because they are objects in the sense of the JSON specification. Other familiar terms for a JSON object might be dictionary, hashmap of associative array. They consist of key-value pairs where each key is a string and each value is one of the values allowed by JSON. Most of the key-value pairs have a special meaning, for example there is a key-value pair that stores the access conrol list.

All objects are part of a tree. For each user, one such tree exists and the root object of the tree is named like the user. Each object can thus be uniquely identified by its path, e.g. alice@wonderlant.lit/social/me.

To store data that can not be expressed in JSON, each object can have an attachment. The attachement is addressed in the same way as the object but is modified using special requests.

2.3. Network topology

FOSP follows the client-server paradigm. The network topology is similar to the SMTP or XMPP network. A provider operates one or more server that handle the domain of the provider. A user can use a client to connect to a server of a provide where he or she is registered and request data.

The server can connect to a server of a different provider to forward request it can not process itself. This is the case when the request concerns an object that is not stored by this server because it belongs to a user of a different provider.

2.4. Messages

The communication between FOSP clients and servers is segmented into messages. There a three different kind of messages: requests, responses and notifications.

To create, alter or delete objects and attachments, a client sends a request to the server. When the request is processed, the server sends a response. If the request concerns an object that the server does not store, it may be forwarded to a server that does store it.

As users and clients might be interested in changes to objects and attachments, they can subscribe to events on objects. When a server makes changes to objects or attachments, it can send notifications to clients that inform about events that happened.

2.5. Bindings

The messages are transported using existing transport protocols. A definition of how FOSP messages are transported through an existing protocol is called binding.

The currently defined binding is the WebSocket binding. An HTTP binding, which will not support all FOSP features, is also worked on.

3. Data structure

FOSP allows users to store structured data in objects and everything else in attachments, which each belong to an object. The objects are all part of a tree and each user owns one such tree.

3.1. Objects

We refer to the basic unit of data, that can be manipulated in the FOSP network, as an object. An object consists of key-value pairs. The constraints for keys and values are the ones described by the JSON specification. Each object has by default specific key value pairs that have a special meaning, for example, the key "owner" contains the identifier of the user who created the object. When transferring objects in messages, the object is serialized according to the JSON specification. In the rest of the document we will refer to a key-value pair of an object either as an attribute or a field of an object.

```
{
 "btime": "2007-03-01T13:00:00Z",
 "mtime": "2008-05-11T15:30:00Z",
 "owner": "alice@wonderland.lit",
 "acl": {
  "owner": {
   "data": [ "read", "write" ],
   "acl": [ "read", "write" ],
   "subscriptions": [ "read", "write" ],
   "children": [ "read", "write", "delete" ]
   },
  "users": {
   "alice@wonderland.lit": {
    "data": [ "read", "not-write" ],
    "acl": [ "read", "not-write" ],
    "subscriptions": [ "read", "not-write" ]
   }
  }
 },
 "subscriptions": {
  "users": {
   "alice@wonderland.lit": {
    "events": [ "created", "updated" ],
    "depth": 1
   }
  }
 },
 "attachment": {
  "name": "hatter.jpg",
  "type": "image/jpeg",
  "size": 140385
 },
 "type": "text/plain",
 "data": "Just plain text"
}
```

Figure 1

3.2. Object attributes

Most attributes of an object have a well defined meaning. The servers have to ensure that the content of these attributes adheres to the specification and that users do not make changes that are not allowed. Figure 1 shows an example of an objects with all the fields described here. Currently we define the following attributes and their values:

The "owner" field is set by the server on creation of the object and contains the identifier of the user who created the object as a string. It determines the ownership of this object and is used when enforcing access rights. The user MUST NOT be able to set the field to a different value.

The "btime" (birth time) field is set by the server to the date of creation of the object. It is saved as a string, formatted according to the ISO 8601 standard and must always be in the UTC time zone. The user MUST NOT be able to set the field to a different value.

Similar the "mtime" (modify time) is set to the current date each time the object is altered and is not to be set by the users directly. It is saved just like "btime". In the future it might be used to allow client side caching similar to ETags in HTTP. The user MUST NOT be able to set the field to a different value.

The "data" field is where the user should save the payload and may contain any valid value. The "type" field should contain a string that describes the content type of the "data" field, similar to MIME types. These two fields are the only fields where the user can save arbitrary data. The server MUST allow the user to store any data in this fields.

Furthermore, there are the three fields "acl", "subscriptions" and "attachment" that contain more complex objects.

The "acl" field saves information about access rights in form of an object. It is read by the server to enforce access control.

The acl object can have the following fields: "owner", "users", "groups", "other"

The value of the "owner" and "other" field is a set of rights

A set of rights is an object where each key identifies a certain scope the right applies to, and the value is a string array of permissions. For example, the key "acl" means that the permissions apply to the acl field of the object. Currently allowed values for the key are "acl", "data", "subscriptions", "children". The permissions can be "read", "write" and "delete". They can also be prefixed with "not-" to revoke a right, which is necessary because rights are inherited. This will be further explained in Section 7.1.

The value of the "users" and "groups" field is an object. In these objects, each key identifies a user or a group and the

related value is a set of rights.

The "subscriptions" field stores information about subscriptions. It is read by the server to determine which users have to be notified on changes to an object.

It contains an object and each key is the identifier of a user.

The value is an object with the fields: "events" and "depth"

The value of "events" is an array of strings which identify an event, the value of "depth" is an integer between -1 and infinity

How this field is evaluated by the server is explained in Section 7.3.

The "attachment" field is only present if this object has an attachment. It contains an object with three fields.

The "name" field contains a string with the file name.

The "size" field contains the number of bytes of the attached file. The value of this field MUST be set by the server whenever a WRITE request changes the size of the file. The value of this field MUST NOT be set by a user.

The "type" field contains a string with the mime-type of the attached file.

3.3. Trees

In FOSP, all objects are part of a tree of objects. Therefore, all objects have a parent object, except for the root object of a tree. Consequently, all objects can have child objects. For each user there exists one tree of objects and the root node of this tree is named like the identifier of the user. The tree of the user is stored on the servers that are responsible for the domain of the provider where the user is registered.

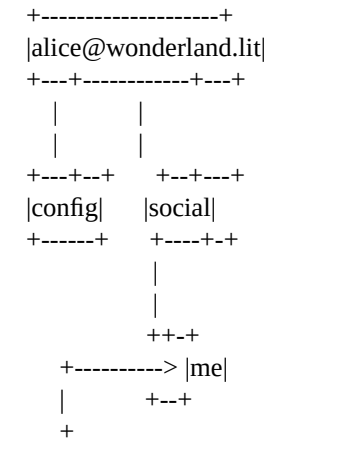

Figure 2

3.4. Attachments

As files, like pictures and documents, are also shared via social networks, FOSP supports saving binary files. For each object, one file can be saved as an attachment. This way, files can be addressed with the same schema and the objects they are attached to can provide the meta-data. If an object has a belonging file, it is extended by a new attribute named "attachment". In this attribute, the "size", file "name", and MIME "type" of the file is saved. The attached file is read and written using special requests.

3.5. Provisioned Objects

Some objects in the tree of a user will be used by the server to obtain configuration options. For example alice@wonderland.lit/config/groups will contain the mappings from users to groups that are valid for the tree alice@wonderland.lit. These objects will be created by the server when the user first registers with it.

4. Network Topology

In the FOSP network, agents are either clients or servers.

4.1. Client

A user connects, using a client, to the server of their provider. We refer to this server as the home server of the user. To manipulate or access data, the user uses the client to send requests.

A client can be any programm that understands the FOSP protocol.

4.2. Server

A server is a software running on a machine on the internet, that processes requests of clients. Each server belongs to one provider that is identified by a domain name. For load balancing purposes, more than one server per domain/provider might be used. To keep the examples simple, we nevertheless assume that there is one server per provider.

A request can act upon a resource. If the resource is not managed by the server the user is connected to, the server will relay the request to the responsible server. Hence, FOSP servers may open connections to other FOSP servers.

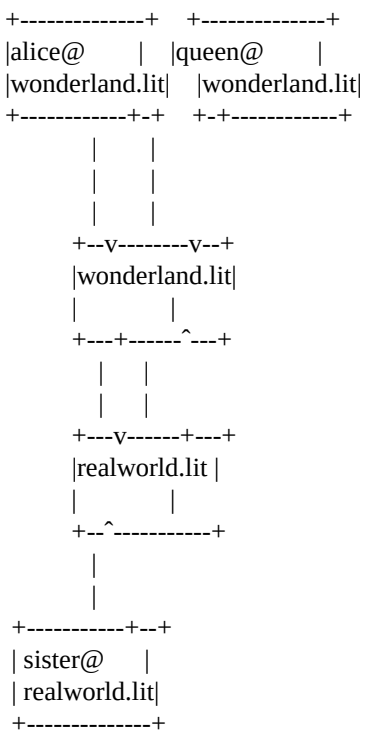
alice@ wonderland.lit
queen@ wonderland.lit
wonderland.lit
realworld.lit
sister@ realworld.lit

Figure 3

5. Messages

Messages are the basic unit of communication in the FOSP network. There are three different types of messages which serve different purposes. How a message is serialized depends on the protocol that is used to transport the message.

5.1. Types

Each message has a specific type, which determines the kind of purpose this message serves. The type is implicitly given in the way that the content of the message determines it´s type. For now there are only three types of messages needed to support all functionality. The types are requests, responses and notifications. Should there be the need for additional types of messages, a new one can be added in later versions of the protocol.

All messages can have headers and a body, similar to HTTP messages and are distinguished by their main attributes. Requests and notifications usually act on, or are emitted from an object or an attachemnt. The identifier of the object or attachemnt is then part of the request. If a request does not act upon an object, the identifier is replaced with an asterisk.

5.2. Request

Requests are sent from clients to servers to authenticate or manipulate objects. A request consists of ...

a request type

optionally an identifier of a resource that it manipulates

The content of the body depends on the type of request. For example, the body of a GET or DELETE request is empty and the body of a CREATE request contains the new object. In general the body is a JSON object, except for the WRITE request that has the byte representation of the file as the body.

The server MUST always responde to a request with a response message. Depending on the request, the server must check access rights and whether a certain object exits, bevor the request can be processed. If the user does not have sufficent rights or an required object does not exist (e.g. a parent object when creating a child), a failure response MUST be sent. When the request is successfully processed, a success response MUST be sent. Depending on the request, it may contain information in its body, like the object that was requested, or an attachment that was read.

The request types are described in the following sections.

5.2.1. Info

There will likely be a request called INFO or OPTIONS to discover the capabilities of the server. However, as of now, it is not yet clear what is needed and therefore the definition is postponed.

5.2.2. Bind

The BIND request is used to authenticate the client. The body MUST contain an object with the "method" attribute that specifies the used authentication method. The possible methods of authentication are not final yet, however every server SHOULD support a simple authentication by name and password. In this case the object also contains a "user" and "password" attribute containing the full user name and the password.

The server verifies that the authentication is successfull and sents a success response or a failed response otherwise. All future request are made in the name of this authenticated user. How the state is kept depends on the transport protocol used. It can for example either be a tied to network connection, or be a cookie supplied in each request.

5.2.3. Get

The GET request is used to retrieve an object. The resource identifier in the request denotes the object that should be returned. The server MUST ignore any content in the body of the request.

If the request was successfull, the object is returned in the body of the response.. Even on success, not the whole object might be returned as the user might have the right to read some attributes but not others.

5.2.4. List

The LIST request is used to discover all child objects of a certain object. The resource identifier specifies the object of which the children should be returned.

If successfull, the response contains an array of names of the child objects.

5.2.5. Create

The CREATE request is used to store new objects on the server. The resource identifier is the desired location for the new object. The body containes the object to store.

On success, an empty success response is returned and the object is stored at the given location.

5.2.6. Patch

The PATCH request is used to update an already existing object on the server. The resource identifier specifies the object that should be updated. The body containes an object with the differences that should be applied to the object.

The differences are merged into the object as follows. If the original object does not have an attribute the differences object has, the attribute is copied to the object. If the original object already has an attribute the differences object has and the new value is , the attribute is deleted from the object. If the original object already has an attribute the differences object has and the value is not null, the attribute is replaced with the new attribute except when the value of the old and the new attribute are both an object. In this case the merging is done on the attribute recursivly as it is on the whole object.

On success, an empty success response is returned.

5.2.7. Delete

The DELETE request is used to remove an object from a tree. The resource identifier specifies the object that should be removed.

If successfull, a succeeded message is returned and the object is permanently removed.

5.2.8. Read

The READ request is used to read the attachment. The resource identifier specifies the object of which the attachment should be read.

The success response body contains the raw bytes of the attachment.

5.2.9. Write

The WRITE request is used to write to the attachment. The resource identifier specifies the object of which the attachment should be written. The body contains the raw bytes of the attachment.

The success response is empty.

5.3. Response

Responses are sent from servers to clients when a request has been processed. A response has ...

a type, either "SUCCEEDED" or "FAILED".

a status code, an integer greater zero.

The body of a response depends on the request that is answered. For a CREATE or DELETE request it would be empty, for a GET request it would be the object that was requested. In case of a READ request it would contain the raw bytes of the attachment.

The status code is explained in Section 5.5.

An FAILED response SHOULD contain an object describing the error that occured. The object should contain a "message" attribute with a string value that explains the error.

5.4. Notification

Notifications are sent from servers to clients when objects change and the client has subscribed to those changes. A notification consists of ...

an event which is one of "CREATED", "UPDATED" or "DELETED".

a resource identifier.

The notifications are sent when a request triggers an event, for example CREATE triggers a CREATED event. If the event equals DELETED then the body of the notification must be empty. Otherwise it should contain the new version of the object.

When and to who notifications are sent is explained in Section 7.3.

5.5. Status codes

The status codes reuse and extend the status codes used in HTTP. They are wildly known and also used in similar fashion in other protocols. The status codes are also divided into the same classes, with the same meaning. Not all status codes of HTTP are used, but all are reserved and MUST NOT be used for custom statuses.

5.5.1. Informal 1xx

The 1xx status codes are not used at the moment. A client MUST ignore any such responses and continue to wait for a final response.

5.5.2. Successful 2xx

The 2xx status codes indicate that the request was successful.

200 OK The request has succeeded.

201 Created The request has succeeded and a new object has been created.

204 No Content The request has succeeded but no content is returned. This can be for example the case when a DELETE request has been sent.

5.5.3. Redirection 3xx

The 3xx status codes indicate that the requested object can be found elsewhere.

301 Moved Permanently The object was moved permanently and is now found at the location indicated in the "Location" header. Moving objects is not yet supported but might be in the future. Clients SHOULD be able to understand this status and in case of a GET, LIST or READ request MAY resent the request with the new location. However, a client MUST never automatically resent the request if the request alters the object, e.g. is a CREATE, PATCH, DELETE or WRITE request.

304 Not Modified If the request did contain a condition that checks for the modification date and the object did not change, the server SHOULD respond with this status code. Conditional requests are not supported yet, but might be in the future.

5.5.4. Client error 4xx

The 4xx status codes indicate that the client made an error.

400 Bad Request The request could not be parsed by the server due to malformed syntax.

401 Unauthorized The client tried to send a request that needs authentication but is not yet authenticated. The client must make a successful BIND request bevor retrying this request.

403 Forbidden The client does not have the necessary permissions. If the server MUST only return this status if the client is already authenticated.

404 Not Found The object the client requested could not be found.

405 Method Not Allowed The request method is not allowed on this object. This status can be returned if a client tries to READ and attachment of an object without attachment.

409 Conflict The request could not be completed due to a conflict with an resource. This status MUST be returned by the server if the client tries to create an object that already exists.

412 Precondition Failed If the request did contain a condition and condition is not satisfied, then the server MUST return this status. Conditional request are not supported yet, but might be in the future. This status can also be returned if a new object should be created but the parent object does not exist.

413 Request Entity Too Large The server might choose to accept only messages smaller than a certain size. In this case, if a client tries to send a larger message, the server CAN respond with this status and drop the request.

5.5.5. Server Error 5xx

The 5xx status codes indicate an error in the server.

500 Internal Server Error The server could not process the request due to an internal error.

501 Not Implemented The server does not support the requested functionality.

502 Bad Gateway The server tried to forward the request, but was unsuccessful. This might be because the authentication failed, the remote server rejected the connection or other reasons. If the remote server can not be reached at all, 504 SHOULD be returned instead.

503 Service Unavailable The server can currently not process any requests but will likly be able to in the near future.

504 Gateway Timeout The server tried to forward the request but could not reach the upstream server at all. This can be a permanent problem and indicate that the requested object does not exist on any remote server at all.

5.6. Forwarding

If a request applies to a resource that is not stored by the home server of the user, the server can forward it to a server that does store it. In this case, the server MUST insert an additional "From" header that contains the full username of the user that sent the request. Similar, when a server sends a response to a forwarded request, an additional "To" header is added. This also applies if a notification is to be sent to a user of a different provider.

6. Bindings

Bindings define how a FOSP message is transported via an existing protocol. Currently a binding for the WebSocket protocol is defined and a binding for HTTP is being worked on.

6.1. WebSocket binding

The WebSocket binding provides a way of sending FOSP messages of WebSocket messages. It was chosen because it is the only bidirectional protocol currently supported in browsers. The client SHOULD NOT use an unsecure WebSocket connection. A server SHOULD NOT accept an unsecure WebSocket connection other then for testing and debugging purposes.

6.1.1. Connection Initiation

To send messages to a server over WebSockets, a connection has to be established first. How the server address, port and URL path are determined for a given domain is discussed in Section 8. In the WebSocket handshake, the Sec-WebSocket-Protocol MUST be set to "fosp".

6.1.2. Format

The FOSP messages is formatted similar to a HTTP request/response.

Also the header definition will be subject to change after determining acceptable characters. For the resource identifier, we will probably refer to the IRI definition in the future.

```
message       = request / response / notification

request       = request-type SP ( resource-id / "*" )
                SP sequence-number CRLF headers [ CRLF body ]

request-type  = "CONNECT" / "AUTHENTICATE" / "REGISTER" / "CREATE"
                / "SELECT" / "UPDATE" / "DELETE" / "READ" / "WRITE"

response      = response-type SP response-status
                SP sequence-number CRLF headers [ CRLF body ]

response-type = "SUCCEDED" / "FAILED"

notification  = event-type SP resource-id CRLF headers [ CRLF body ]

event-type    = "CREATED" / "UPDATED" / "DELETED"

headers       = *( header CRLF )

header        = header-key ":" SP header-value

header-key    = ALPHA *( ALPHANUM / "-" ) ( ALPHANUM )

header-value  = 1*( ALPHANUM / "/" / "-" / "<" / ">" / "="
                / "+" / "_" / ";" / "!" / "~" / "*" / "." )

resource-id   = user-name "@" domain [ path ]

user-name     = ALPHA *( ALPHANUM / "_" / "-" / "+" / "." )
                ( ALPHANUM )

domain        = domain-part *( "." domain-part ) [ "." ]

domain-part   = ALPHANUM / ( ALPHANUM *( ALPHANUM / "-" / "_" )
                ALPHANUM )

path          = "/" / 1*( "/" path-part )

path-part     = 1*( ALPHANUM / "\" / "." / "+" / "*" / "-" / "_"
                / ";" / ":" / "!" / "~" / "=" / "<" / ">" )

ALPHANUM      = ALPHA / DIGIT
```

Figure 4

6.1.3. Serialization

Messages are serialized into a blob of bytes and sent inside a single WebSocket message. Most of the message is text in UTF-8 encoding, only the body is either a valid JSON object, UTF-8 encoded, or raw bytes. The second case occurs when an attachment is up- or downloaded. The beginning and end, e.g. the length of a message must be determined from lenght of the WebSocket message.

When a message contains binary data that is not UTF-8 encoded text, e.g. a WRITE request, a binary WebSocket message MUST be used. Otherwise, when the whole message is valid UTF-8 encoded text, a text WebSocket message CAN be used.

6.2. HTTPS binding

The HTTPS binding is developed because it might be simpler for clients to process large up- or downloads via HTTPS. However, it does not support all of FOSPs functionality because notifications can not be sent over HTTPS (at least over HTTP/1.1). A client SHOULD NOT issue FOSP requests over plain HTTP. A server SHOULD NOT allow HTTP based requests other then for testing and debugging purposes.

6.2.1. Mapping

Each FOSP request is transported as an HTTP request. The request identifier is used as the HTTP method. The HTTP request-URI MUST be the absolute URI of the FOSP object. The HTTP headers are the FOSP headers. The HTTP body is the FOSP body.

6.2.2. Session management

To support authenticated requests, a FOSP server that supports the HTTP binding MUST return a cookie in a success response to a BIND request. With this cookie, the client MUST be able to make authenticated requests.

7. Policies

Besides implementing the technical specification of the protocol, the server has to enforce a set of policies. These policies describe how the server interprets the objects to enforc access control and send notifications for subscriptions. They also define constraints on certain attributes of objects and constraints on whether or not a message should be forwarded.

7.1. Access control

As stated in Section 3.2, each object can save access control information. A valid entry in the access control list consists of the identifier of a user or a group of users and a list of rights. Manipulation of different parts of the object can require different rights, for example, altering access control information requires the "write" right in the "acl" key. Because the objects are part of a tree, we can use inheritance to set access rights for a whole subtree by setting the appropriate access control information on the root of the subtree. This way the rights on an object are the sum of the rights on this object and all it´s ancestors. To still be able to have less rights on a child object, rights can be prefix with "not-" to explicitly remove a right, for example "not-write". The server must enforce access control for an object in the following way.

1. Check if the right in question is set on the current object
2. If it is set positive (e.g. not prefixed with "not-") grant access
3. If it is set negative (e.g. prefixed with "not-") deny access
4. If is is not set, go to the parent object and repeat from step 1.
5. If the current object is already the root object, deny access

7.2. Groups

To be able to set rights for a group of people, the user must first be able to define groups and store them at a well known place on the server. Each group is only valid for the tree that it is defined in. The server MUST look for group definitions in objects at the location "<user@domain>/config/groups". Each object underneath this object can contain a group definition and the group name is the name of the object. To be a valid group definition, the object MUST contain an object in its "data" attribute and this inner object MUST contain a "users" attribute with an array value. The array then contains the full names of all users in this group.

7.3. Subscriptions

Similar to the access control information, subscriptions can be set on objects so that a user will be notified when changes occur. A subscription consists of an identifier of a user, a list of events to subscribe to and a "depth". The depth is used to subscribe to events from all child objects to a certain depth. When a change happens on an object, the server must use the following algorithm to determine which users must be notified.

1. Read the "subscriptions" from the current object.

2. For each subscription: If the event that occurred is in the list of subscribed events and the distance of the current object to the object where the event occurred is smaller or equal to the "depth" of the subscription or the "depth" is equal to "-1", add the user of this subscription to the list of users that should be notified. The distance between two objects is 0 if they are the same object, 1 if one is the parent of the other and so on.

3. Go to the parent of the current object and repeat from step 1. unless the current object is the root object.

An important aspect to consider when sending notifications that include the new version of the object, is that every user who will be notified might be allowed to only see different parts of the object. Therefore, the server has to calculate the view of the object per user to prevent leaking of data a user might not be allowed to see.

7.4. Attachments

As explained in Section 3.4, each object can have a file as attachment. However, attachments will likely not be stored together with objects, but in a storage that is more suitable for files. Depending on how the server implements operations on attachments, it is possible that there exists an attachment for an object in the storage but the object itself is missing the "attachment" field. In any case the behavior of the server should be consistent. Hence, if there is no attachment field in the object then the attachment should be deleted and an attachment should only be readable if there is an attachment field in the object.

7.5. Authentication

Most of the operations a user performs in FOSP probably require some access rights. To enforce these access rights, a user must be authenticated and therefore be able to authenticate with the server. As a server might forward requests on behalf of a user, the remote

server must also be able to verify that the server is authorized to forward requests for the user. Therefore, servers must also authenticate each other.

7.5.1. Users

When a client connects to a server it can use the BIND request to authenticate itself. The most basic way to do this is just to provide a username and a password, which SHOULD be supported by any server. In the future, additional authenticaten methods will be added.

7.5.2. Servers

When a server opens a connection to another server, the authentication can be done using DNS. The connectiong server must provide a domain name it wants to be authenticated for. The accepting server then does the server discovery process described in& nbps;Section 8.1 and compares the resolved IP address to the address of the connecting server. If the IP addresses match, the authentication is successfull.

7.6. Registration

The registration of a new user account is no longer part of the FOSP protocol. In general, out-of-band registration is prefered. It can either be done by the user via a web formular or the accounts can be provisioned, for example in a coporate network from a directory server. In the future, a possibility of creating and account via an CREATE request might be specified.

7.7. Message forwarding

In general, a server should accept forwarded messages from another server. However it MUST authenticate the remote server first, before processing any other requests or sending any notifications.

A server does not need to accept forwarded requests from other servers, if it shouldn´t be part of the federated network. This allows FOSP to be used in cooperate environments or other closed environments where federation with the outside world is not allowed.

In any case, a server MUST never accept a forwarded request if the user of the request is not on the domain of the server that forwarded the request. For example, if server A, that is authenticated for domain "example.net", authenticates to server B and then forwards a request of user "alice@wonderland.lit", server B MUST close the connection. The user from which the request originates MUST be

specified in the "From" header of the request.

8. Discovery

To be able to connect to a server, the client has to first determine the correct address of the server. The server is determined using the domain name of the user name. There are two different ways of determining the server address, a DNS based one and by retrieving a JSON object from a "well-known" location. The second approach is added as web application clients are unable to perform DNS lookups directly.

8.1. DNS

The discovery via DNS uses SRV records. Unfortunately, the current format of SRV records is limited as it only includes one transport protocol. DNS based discovery is therefore OPTIONAL until we are certain that the defined record format is acceptable.

To discover a WebSocket based FOSP server for wonderland.lit, a DNS lookup for _fosp._wss._tcp.wonderland.lit with the record type SRV hast to be performed. Similar, to discover a HTTP based FOSP server, _fosp._https._tcp.wonderland.lit has to be resolved.

8.2. Well known

When DNS lookups are not possible, the server address can also be discovered by retrieving a JSON object. In accordance with [RFC5785] the object MUST be located at https://<domain>/.well-know/fosp. A client SHOULD NOT accept this file over plain HTTP.

This response, if successful, MUST contain an JSON object and SHOULD have the "Content-Type" header set to "application/json". The object MUST either contain an "https-tcp" attribute with a string value of an valid HTTP URL or a "wss-tcp" attribute with a string value of a valid WebSocket URL or both.

9. References

[refs.rfc2119]
Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, BCP 14, March 1997.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, April 2010.

Author´s Address

Felix K. Maurer

Email: felix.maurer@student.kit.edu