Network Working Group
Internet-Draft
Expires: September 2, 2014

F. Maurer
March 2014

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 independed providers. It was designed to be the most simple solution that combines all these features.

Status of this Memo

Table of Contents

1. Introduction

1.1. Purpose

TODO

The federated object storage protocol tries to solve three problems:

1. How to store data in a network and support standart operations on it.

2. How to enforce access control on the data, for useres from different hosts, without central authentication.

3. How to notify users when data is added, removed or has changed.

Additionally we want to keep following constraints:

1. The protocol needs tp be data agnostic and should support structured, unstructured and metadata.

2. The protocol must be as simple as possible, it must be easy to implement and programms that use it should be easy to deploy.

1.2. Overview

The FOSP protocol is mostly a request/response protocol. A user connects to the server of his or her provider, using a client and then both can exchange messages. Request messages are sent from the client to the server to manipulate or retrieve data. Servers respond by sending a response message.

The information units stored on the server have a well defined structure and we call them objects. They are furthermore organized into trees. Each user is identified uniquely by a name and the domain name of his or her provider, similar to an Email address. For each user there exists a tree of objects, stored on the server(s) of his or her provider.

1.3. 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].

1.4. Requirements

FOSP builds on existing protocols and data formats. The transport protocol for FOSP messages currently is the WebSocket protocol, tough other options may be added in the future. Objects are serialized into the JSON notations. Besides these technical dependencies, FOSP is inspired by WebDAV and XMPP and has similarities to LDAP.

1.5. 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 DNS domain name.

server A FOSP server stores the data of users of a certain provider. Multiple servers can be deployed to deliver the data of a provider to implement fault tolerance and load balancing.

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.

2. Network Topology

In the FOSP network, agents are either clients or servers. Each server belongs to one provider that is identified by a domain name. For load balancing purposes, more than one server per domain might be used. To keep the examples simple, we nevertheless assume that there is one server per provider and use the term provider and server interchangeable. 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 sends requests. A request can act upon a resource. If the resource is not managed by the server the user is con- nected to, the server will relay the request to the responsible server. Hence, FOSP servers may open connections to other FOSP servers.

3. Messages

Messages are the basic unit of communication in the FOSP network. There are three different types of messages which serve different purposes. After being serialized, each message is transported over a WebSocket connection inside of one WebSocket text message or binary message.

3.1. Types

Each message has a specific type, which determines the kind of purpose this mes- sage 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 sup- port 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.

3.1.1. 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

a sequence number

The request type is one of the following ...

CONNECT	used to initiate a connection
REGISTER	used to create a new FOSP user
AUTHENTICATE	used to authenticate
SELECT	retrieves an object
CREATE	adds a new object
UPDATE	changes an existing object

DELETE   removes an object and all its children

LIST   returns a list of children of an object

READ   download the attached file of an object

WRITE   upload a file as an attachment to an object

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

3.1.2. 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.

a sequence number which must match the sequence number of the request it responds to.

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

3.1.3. 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 CRE- ATE 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.

3.2. Format

WARNING: To simplify the definition for the moment, only ASCII characters are used. This WILL change as the protocol itself mandates the use of Unicode and the UTF-8 encoding and international user and resource names MUST be supported

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 1

3.3. Serialization

Messages are serialized into a blob of bytes. Most of the message is text in UTF- 8 encoding, only the body is either a valid JSON object, UTF-8 encoded, or not interpreted at all. The second case occurs when an attached file is up- or down- loaded. The beginning and end, e.g. the length of a message must be determined from lenght of the WebSocket message.

4. Data structure

4.1. 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 transfering objects in messages, the object is serialized according to the JSON specification.

4.2. 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 where the user is registered.

4.3. 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 attach- ment. 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 attributes 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.

4.4. Standart Fields

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 4.5 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 "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. Users should not be able to change its content. It was added because it´s an information that is usually of interest to users.

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 will be used to allow client side caching similar to ETags in HTTP.

e recommend that users should not be allowed to add or alter arbitrary fields except for the "data" and "type" field. 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.

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

The "acl" field saves information about access rights. 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 array of strings where each string identifies a cer- tain right. For example, the string "read-acl" stands for the right to read the "acl" field.

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 saves 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

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 "mime-type" field contains a string with the mime-type of the at- tached file.

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

Figure 2

4.5. Standart Objects

Some objects in the tree of a user will be used by the server to obtain configurations. For example alice@wonderland.lit/config/groups will contain the mapping from users to groups.

5. 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 enforces 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.

5.1. Access control

As stated in the previous section, 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 re- quire different rights, for example, altering access control information requires the "acl-write" right. 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-acl-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

5.2. 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.

5.3. Attachments

As explained in Section 4.3.3, 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 stor- age 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.

5.4. Authentication and Registration

When a client connects to a server, it has to provide credentials to the server, so that the user of the client can be authenticated. These are send in the body of the AUTHENTICATE request and can simply be an object containing the name and the password of the user. The schema of authentication is not limited thought as the body can contain arbitrary structured data. Server to server authentication is done using the DNS. When a server receives a connection from another server it verifies that the domain of the connecting server resolves to the same IP address the connection comes from.

The REGISTER request should be supported by all servers, but can be dis- abled to prevent automatic creation of accounts by autonomous programs. It is also not required that the REGISTER command is the only way to create an ac- count on a server. For example, there could be sign up forms on websites or a server of a company could be connected to a directory service and fetch the login informations from there.

5.5. Message forwarding

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 co- operate 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 authoritative for domain "example.net", authenticates to server B and then forwards a request of user "alice@wonderland.lit", server B must close the connection.

6. Connection Initiation

First, the agent that wants to open a new connection has to find out which server to connect to and on which port. The server responsible for a certain domain is the server with the IP address that is assigned to the DNS A record of the do- main name and the port is 1337. In the future, a better approach of discovering the server will be necessary and probably done by using DNS SRV records.

After opening a connection to a server, the initiator first sends a CONNECT request that has to be answered with a SUCCEEDED response if the connection is accepted. The body of the request contains information about the supported version of the protocol and could in the future be used to negotiate more pa- rameters of the connection. The response, if successful, contains the version the server supports and could also be used to advertise additional services provided by the server.

If the connection is opened by a client to a server, a new user can then be registered with the REGISTER request. Finally, the initiator has to authenticate to the server he connects to. The specific mechanism of authentication depends on whether a client connects to a server or a server connects to another server.

7. References

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

Author´s Address

Felix K. Maurer

Email: felix.maurer@student.kit.edu