The Prototype Protocol Flow

This is the description of the protocol flow I use in my prototype at http://host.clproject.net/

General notes

- Instead of XRD I use a slightly modified variant of JRD as it is simpler to parse.
- The Host contains three resources of Bob: a basic, a medium and a detailed profile.
- The only access conditition right now is a check on the username the requesting party provides at the AM. This could be an OpenID login in a real world scenario.
- many things are just inventions of my own where I thought it might work that way. These need discussion and proper specification.

Step 1: Introduction of Host to the AM

Step 1 starts when the user enters the location of the AM to be used for.

1.1: Obtaining the AM metadata

In order to start the process the Host needs to know various URI endpoints of the AM. These are retrieved by a hostmeta lookup by the Host (here is the live AM hostmeta). Technically it's a GET on the hostmeta URL and this is what it returns:

GET http://am.clprojects.net/.well-known/hostmeta

```
{
    "property": [
        {
            "http://uma/am/title": "Cop Monkey"
        }
    ],
    "expires": "2011-01-30T09:30:00Z",
    "link": {
        "http://uma/host/token_uri": [
            {
                "href": "http://am.clprojects.net/host/token"
            }
        ],
        "http://uma/host/user_uri": [
            {
                "href": "http://am.clprojects.net/host/authorize"
            }
        ],
        "http://uma/host/resources": [
            {
                "href": "http://am.clprojects.net/host/resources"
            }
        ],
        "http://uma/requester/user_uri": [
            {
                "href": "http://am.clprojects.net/requester/authorize"
            }
        ],
        "http://uma/requester/token_uri": [
            {
                "href": "http://am.clprojects.net/requester/token"
            }
        ]
    },
    "subject": "http://am.clprojects.net/"
}
```

We see various endpoints here:

- a OAuth 2.0 user_uri endpoint for the Host to use (http://uma/host/user_uri)
- a OAuth 2.0 token_uri endpoint for the Host to use (http://uma/host/token_uri)
- a OAuth 2.0 user_uri endpoint for the Requester to use (http://uma/requester/user_uri)

- a OAuth 2.0 token_uri endpoint for the Requester to use (http://uma/requester/token_uri)
- a UMA resource registration endpoint (http://uma/host/resources). This is my invention

The Host stores all these endpoints for later usage.

1.2: Host registers Bob's resources at the AM and obtains client_id

In order for the AM to be able to manage Bob's resources it needs to know what these resources are. Instead of entering the URIs of them manually I invented some registration step to do so automatically.

For this the Host crafts a JRD document containing the resources identified by their resource URI and a title. The example in the prototype looks as follows:

Resource Document Example

```
{
    "subject" : "http://host.clprojects.net/profiles/bob",

    # we copied this from the hostmeta JRD so we can skip one lookup here
    "property":
    [
      { "http://uma/host/title" : "UMA Example Host" },
    ],
    "link" : {
        # our namespace defining a protected resource
        "http://uma/am/resource" : [
            # we have three protected resources as links
            {
                'href' : 'http://host.clprojects.net/profiles/bob.basic',
                'title' : 'Basic Profile',
            },
            {
                'href' : 'http://host.clprojects.net/profiles/bob.medium',
                'title' : 'Medium Profile',
            },
            {
                'href' : 'http://host.clprojects.net/profiles/bob.detail',
                'title' : 'Detailed Profile',
            },

        ]
    }
}
```

This document is sent to the `http://uma/host/resources` endpoint of the AM which we retrieved via the hostmeta lookup. This is done via a POST request.

The AM then stores this information and assigns it a `client_id` which then is returned to the Host. This is done in the response of the POST request. The document is JSON formatted and looks liks this:

```
{
    'client_id' : '622f5656-7211-4315-a69f-24fde3d65732',
}
```

Now the AM knows about the resources of the Host and the Host has a `client_id` which can be used in an OAuth 2.0 web server flow.

1.3: Host obtains an Access Token of the AM via an OAuth 2.0 Web Server Flow

The Host initiates an OAuth 2.0 Web Server Flow with the `client_id` it got to the `http://uma/host/user_uri` endpoint from the AM hostmeta. The user (Bob) will be redirected there and the AM will then let Bob configure his resources (which are known because of the `client_id`). This is not in the scope of the specification though. In our example Bob can configure a username per resource which will then later be allowed to access it.

The user will then be redirected to the Host with an OAuth `code` parameter which the Host will then exchange for an access token via the `http://uma/host/token_uri` endpoint it knows from the AM hostmeta.

After that the initialization flow is finished and the Host has an Access Token (we call it *Host Access Token*) from the Authorization Manager to be used later in step 3.

Step 2: Requester obtains an access token to access the protected resource

Now Mary on the Requester wants to access Bob's resources. To do that properly the Requester needs a *Requester Access Token* from the AM which it can send to the Host.

2.1: Requester tries to access protected resource and obtains the location of the AM

In order for the Requester to ask the AM for an Access Token it needs to know which AM to use. To find that out it will send a request to the protected resource (e.g. http://host.clprojects.net/profiles/bob.basic). As no access token is provided, the Host will answer with

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: user_uri=http://am.clprojects.net/requester/authorize,token_uri=http://am.clprojects.net
/requester/token
```

This is probably not valid OAuth and this probably needs discussion. The two URIs are known once again from the AM's hostmeta file (the resource endpoints) which the Host assigned on behalf of Bob to these resource URIs in step 1.

Now the Requester knows the OAuth endpoints of the AM and can start an OAuth flow to obtain the access token.

2.2: Requester start UMA OAuth flow to obtain an access token

The OAuth flow being used is a little different from the normal OAuth 2.0 Web Server Flow but very similar. The main difference is that the `client_id` is optional (because the Requester hasn't registered yet with the AM. This needs discussion though) and because it can provide a `callback` parameter.

The Requester crafts a redirect of Mary to the AM with the following parameters:

type

This is `uma_web_server`

scope

This contains the resource uri the Requester wants to access. This apparently needs discussion.

redirect_uri

The normal OAuth `redirect_uri` parameter.

callback_uri

In some cases the access to the resource is not granted directly maybe because the resource owner first needs to be asked by SMS or email. In such cases the OAuth flow redirects back to the Requester with a code but the code is not yet valid and will return some error indicating that the access is still pending. The Requester might poll in such cases or provide this callback URI to which the AM will then send the access token once it has been granted (or an error if access was denied).

This hasn't been implemented in this case though and thus it's here only for discussion.

The request

All of these parameters (plus the other optional OAuth web server flow parameters) are then crafted into the `user_uri` obtained in 2.1 and then the user is redirect to the AM.

2.3: AM checks if access is being granted

Many methods are possible here and UMA will provide some (optional) claims mechanism in it's spec. There are many things the AM can check itself though, e.g. it could let the requesting party log in via OpenID to obtain it's identity and then check with configuration the resource owner did in Step 1. Or it could simply ask the resource owner via SMS or E-Mail as explained above.

The prototype simply asks for a username and if this matches the username Bob configured earlier for this resource, then access is granted. Otherwise it is not.

The AM will thus simply return the normal OAuth flow without using any callbacks and such and the Requester will then try to exchange the code with an access token which either works or not.

The Requester will then have a `Requester Access Token` from the AM which it can use to access the resource on the Host.

Step 3: Requester accesses the protected resource

In the last step the Requester accesses the protected resource by taking it's `resource_uri` and adding `?oauth_token=<requester access token>` to it.

In this prototype the only check is whether an access token is present or not but in a complete example the following would happen in order to check if the access token is valid or not:

#. The Host receives the Requester Access Token via the call
#. The Host contacts the AM with it's own Host Access Token and asks whether the Requester Access Token is valid
#. The AM checks it's database and answers yes or no.
#. The Host either grants access or not depending on the answer (and it's own checks maybe).

This is not implemented in the prototype though and you might refer to the SMART implementation to learn more about ideas on how to implement claims and such.

Further reading

- You can find the source code of this prototype at http://bitbucket.org/mrtopf/uma