Computer Networks 57 (2013) 2233–2249

Contents lists available at SciVerse ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Integration of the OAuth and Web Service family security standards

Elena Torroglosa-García[a,*], Antonio D. Pérez-Morales[b], Pedro Martinez-Julia[a], Diego R. Lopez[c]

[a] *Department of Communication and Information Engineering, University of Murcia, 30100 Murcia, Spain*
[b] *RedIRIS, The Spanish National Research and Education Network, 41012 Seville, Spain*
[c] *Telefonica I+D, 28050 Madrid, Spain*

ARTICLE INFO

Article history:
Available online 20 April 2013

Keywords:
Security Token Service
Authentication
Authorization
WS-Security
OAuth
GEMBus

ABSTRACT

There are more and more scenarios requiring the transparent integration of heterogeneous security services in order to facilitate application development, simplify deployment and provide a seamless user experience. One of the most common use cases occurs when resources make use of OAuth to provide a simple and flexible way to authorize clients in order to access protected resources. But different OAuth implementations normally use distinct types of authorization grant and access tokens. This heterogeneity can be tackled by leveraging on WS-Trust, which is especially intended to offer integration mechanisms among services that implement WS-* specifications. By integrating these mechanisms it is possible to reduce the complexity supported by the OAuth Authorization Server (AS), so easing the interoperability through the delegation of the issuance and validation processes. This work also proposes a solution to cover the needs of WS-Trust clients which intend to use OAuth resources.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

Nowadays, the number and variety of services that can be found on the Internet is growing, and most of them need to be secured in order to not reveal information to those people not authorized to access to it. Much of this information is related to private user data, so the security takes on an important role when it deals with services accessing private information. There are many scenarios requiring the transparent integration of heterogeneous security services. This integration eases the application development, as well as simplifying deployment and providing a seamless user experience.

Traditional services are based on top of the SOAP protocol because they are easy to use, their interfaces are defined by a contract (using WSDL) and there are many development tools that help to build SOAP services. Moreover, many specifications were created in order to secure this type of services, such as WS-Security and WS-Trust (all of them members of the WS-* family of web service specifications). WS-Security incorporates security features in the header of a SOAP message and can be used in conjunction with other Web service extensions and higher-level application-specific protocols to accommodate a wide variety of security models and security technologies. WS-Trust, in turn, provides extensions to WS-Security. WS-Trust deals with the issuance, validation, renewing and cancelation of security tokens that are used to provide security using the mechanisms created by WS-Security.

In recent times, the services based on the REST protocol are gaining strength, because they are lightweight and easy to develop (specific tools are not needed) and widely supported by the growth of social networks. To secure REST services, one of the latest security protocols that has been created is OAuth. It is an authorization protocol and is oriented to the interconnection and integration of service

* Corresponding author. Tel.: +34 868884644.
E-mail addresses: emtg@um.es (E. Torroglosa-García), antonio.perez@rediris.es (A.D. Pérez-Morales), pedromj@um.es (P. Martinez-Julia), diego@tid.es (D.R. Lopez).

1389-1286/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.comnet.2012.11.027

APIs used by Facebook and Twitter, the two major social networks. These social networks make use of the OAuth protocol to provide a way to authorize clients for access to protected information.

Both OAuth and WS-Trust (along with WS-Security) are the main protocols used today to provide security functions to web services. However, different services using different security mechanism need to be integrated in a transparent way. This integration is not trivial because different services may use different and incompatible security mechanisms. One of the most common use cases for such security services is found in the services making use of the OAuth protocol to provide a simple and flexible way to authorize clients to access the protected information (resources). Due to the fact that different OAuth implementations normally use distinct types of authorization codes and access tokens, the integration between these implementations is not an easy task.

However, this heterogeneity can be tackled by leveraging on WS-Trust, which offers integration mechanisms between services which implement WS-* specifications. Thus, this work proposes a combined solution for this type of scenario. In our proposal, the integration among services that use different protocols (SOAP and REST) and different security mechanisms is achieved through the integration of the mechanisms proposed by OAuth and WS-Trust. Also, by integrating these mechanisms it is possible to reduce the complexity supported by the OAuth Authorization Server (AS), so easing the interoperability by delegating the issuance and validation processes to the WS-Trust mechanisms which are more mature and are able to deal with a wide range of security tokens. In addition, this work also proposes a solution for the integration of WS-Trust clients which intend to use OAuth protected resources. It closes the cycle and opens a wide range of authorization possibilities to current and future web services.

The remainder of this paper is organized as follows. Section 2 introduces the current situation in the context of the integration of security protocols, including the pillars of this study such as the Web Service family specification and the OAuth Protocol. Section 3 exposes the interoperability problem between WS-* and OAuth environments and offers two different points of view for integration scenarios. Section 4 explains the context of the prototype implementation based on the GEMBus Project and the OAuth Authorization Service. The explanation stresses the description of the Security Token Service (STS) (Section 4.2), which is the particular security service found in the GEMBus architecture. Later, using the implemented solution detailed in the previous section. Section 5 describes the validation of the proposed integration models through the analysis of two scenarios. Section 6 concludes the paper and gives some indications of the future work.

2. Background

There is a large heterogeneity between services due to the boom of new web services and different authentication and authorization protocols. There is a need to move towards integration scenarios that allow resource sharing and improve the user experience, providing mechanisms which ensure the privacy of the users.

There are different alternatives for achieving this goal, for example, translation protocols such as WS-Trust and WS-Security and also generic mechanisms such as SASL and GSS-API which allow the applications to abstract from the implementations of specific security protocol. The last two mechanisms have the drawback of only partially supporting the authorization. They are intended rather for authentication. On the other hand, OAuth is an authorization protocol that includes integration mechanisms for using authentication statements such as SAML tokens.

2.1. Web Services family specification

2.1.1. WS-Security

Web Services Security [1] is a communication protocol that provides the means for applying security to web services. It is a member of the WS-* family of web service specifications and is developed by a committee within OASIS [2]. The last revision of the standard (version 1.1) was published in 2006.

The protocol is officially called WSS and it is associated with the following approved specifications: WS-Trust, WS-SecureConversation and WS-Policy, and is mainly focused on the use of XML Signature and XML Encryption to provide end-to-end security.

The specification is intended to provide a flexible set of mechanisms that can be used to construct a range of security protocols. It defines a flexible and feature-rich extension to SOAP [3] to apply security to web services, specifying how integrity and confidentiality can be enforced on messages and also the communication of various security token formats, such as SAML [4], Kerberos [5] and different types of certificates such as X.509 [6]. Some of these standards are described below.

2.1.1.1. SAML. Security Assertion Markup Language (SAML) is an open standard developed and approved by the OASIS Security Services Technical Committee. It makes use of XML language to exchange authentication and authorization data between security domains, usually between an identity provider and a service provider. SAML language ([7]) defines the exchange of authentication, attributes and authorization decisions statements. Authentication statements inform the service provider that the user has successfully authenticated with the identity provider using a specified authentication method. Attribute statements provide name-value pairs with information associated with the subject that can be used by a service provider to make access control decisions. Authorization decision statements assert that the subject has been authorized to perform a certain action on a specific resource.

To comply with the requirements detailed in the WSS standard, the producer sends a SAML assertion expressing data related to the authentication, including a valid audience, and restricted to the resource it is addressed to, through a SAML condition element containing a unique URI that identifies the resource. The assertion includes a statement that specifies the method of relayed trust that must be used to evaluate the assertion, through a specific value in the SAML constructor to identify the subject confirmation method. This value is the URI in the eduGAIN namespace: urn:geant:edugain:reference:relayed-trust. The SAML assertion(s) is received from the IdP as evidence of this confirmation process, as part of the SAML element SubjectConfirmationData.

2.1.1.2. Certificates. In cryptography, a digital certificate (also known as a public key certificate or identity certificate) is an electronic document by which a trusted third party (certification authority) ensures the binding of a public key with an identity using a digital signature. The identity may make reference to an individual or entity and includes information such as the name of a person or an organization, their address, and so forth. The certificate can be used to verify that a public key belongs to an individual.

For a certificate to be used to identify and authenticate, the use of the private key (which only the owner knows) is necessary. The certificate and the public key are considered non-sensitive information that can be distributed to third parties perfectly. So a certificate cannot be used as a means of identification, but it is an essential piece in the protocols used to authenticate the parts of digital communication, to ensure the binding of the public key and the identity.

X.509 [6] is an ITU-T standard [8] for a public key infrastructure (PKI) [9] and Privilege Management Infrastructure (PMI). It was begun in association with the X.500 standard, and specifies, amongst other things, standard formats for public key certificates, certificate revocation lists, attribute certificates, and a certification path validation algorithm.

WS-Security defines a X.509 Certificate Token Profile [10] that describes the syntax and processing rules for the use of the X.509 authentication framework with the SOAP Mesxsage Security specification. This specification defines the transport of the certificated embedded in a `<wsse:BinarySecurityToken>`

2.1.2. WS-Trust

WS-Trust [11] is a WS-* specification and OASIS standard that provides extensions to WS-Security, specifically dealing with issuing, renewing and validating security tokens, as well as how to establish, assess (the presence of) and broker trust relationships between participants in a secure message exchange. WS-Trust defines the concept of the Security Token Service (STS) as a web service that issues security tokens as defined in the WS-Security specification. Besides, the standard specifies the formats of the messages used to request security tokens and the responses to those messages and the mechanisms for a secure key exchange.

A STS is a Web service that issues, renews and validates security tokens based on evidence that it trusts. These security tokens are defined in the WS-Security specification. To communicate trust, a service requires a proof such as a signature, or the proof of knowledge of a security token (or a set of them). The token generation process can be carried out by the service itself or can rely on a separate STS. In this case, the separate service issues the security token with its own trust statement.

Fig. 1 shows the three main entities in the specification. The requester -Requestor- represents a service or client that wants access to a Web Service. This Web Service may require that the incoming message proves a set of claims. If the requester does not have the necessary token to prove the claims required by the service, it can contact appropriate authorities and request the information needed with the proper claims. This authority is STS, which may require its own set of claims for authenticating and authorizing the request from the requestor.

The WS-Trust specification defines the formats of the messages to request and reply for the issuance, the validation and the renewal of security tokens. The issuance process makes use of the `RequestSequrityToken` request that includes details about the type of the token required, the type of request and other information that might be needed, such as user credentials or other attributes. The reply provides the new security token and other related information, such as the type and the request context.

In the case of the renewal binding, the request is similar to the issuance one. The main difference is the required inclusion of `RenewTarget` element that identifies the token being renewed and may contain a reference to the token or, directly, the token to be renewed. In response to this request, the consumer receives the fresh token from the STS.

The validation binding shows a similar pattern to the previous messages. In this case, it is necessary to include the `ValidateTarget` element that typically contains the reference to the token or could also carry the token to be validated directly. After STS has evaluated the token, it replies to the consumer with the token status, including a specific code (valid or invalid) and, optionally, the reason as a human-readable text.

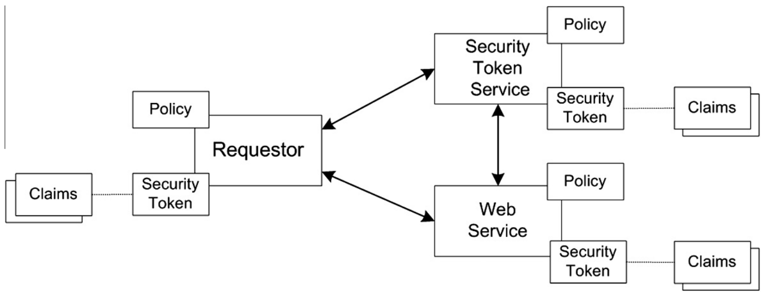

Fig. 1. WS-Trust security token service scheme.

2.2. The OAuth protocol

OAuth is an authorization protocol that enables a third-party application to obtain limited access to a service, either on behalf of a resource owner or by allowing the third-party application to obtain access on its own behalf.

In the traditional client–server authentication model, the client requests an access restricted resource (protected resource) from the server by authenticating with the server using the resource owner's credentials. In order to provide third-party applications access to restricted resources, the resource owner shares its credentials with the third-party. This creates several problems and limitations. Third-party applications are required to store the resource owner's credentials for future use, so the compromise of any third-party application results in a compromise of the end-user's password and all the data protected by that password. Resource owners cannot revoke access to an individual third-party without revoking access to all third-parties, and must do so by changing their password. Besides, third-party applications gain overly broad access to the resource owner's protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.

The OAuth protocol addresses these issues (and more) by introducing an authorization layer and separating the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server, and is issued a different set of credentials than those of the resource owner. Instead of using the resource owner's credentials to access protected resources, the client obtains an access token (a string denoting a specific scope, lifetime, and other access attributes). Access tokens are issued to third-party clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server.

2.2.1. Roles

The OAuth protocol defines four role. The resource owner which is an entity capable of granting access to a protected resource. The client, which is an application making protected resource requests on behalf of the resource owner and with its authorization (it could be an application executed on a server, desktop or other devices). The authorization server (AS), which is the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization. And finally, the resource server (RS) is the server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens. These roles are used in the OAuth architecture inside the flows depicted in Fig. 2.

2.2.2. Architecture

The OAuth architecture is very simple and its definition is based on the abstract protocol flow as is depicted in Fig. 2, which describes the interaction between the roles. As can be seen, the steps are as follows: first, the client requests and obtains authorization from the resource owner to access a protected resource. The client receives an authorization grant which is a credential representing the resource owner's authorization. Then, the client requests an access token by authenticating with the authorization server and presenting the authorization grant. When the authorization server receives the request, it authenticates the client and validates the authorization grant, and if it is valid, issues an access token to the client. The client then requests the protected resource from the resource server and authenticates by presenting the access token. The resource server receives the access token, validates it, and if it is valid, serves the request.

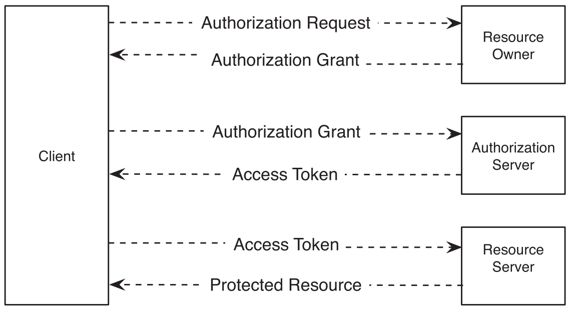

Fig. 2. Abstract protocol flow.

2.2.3. Access token request

To obtain an access token, the client makes a request to the token endpoint by adding the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body:

- `grant_type` (required): the value MUST be set to "authorization_code".
- `code` (required): the authorization code received from the authorization server.
- `redirect_uri` (required, if the "redirect_uri" parameter was included in the authorization request).

For example, the client can make an HTTP request using TLS as shown in Table 1.

2.2.4. Extension capabilities

For the definition of new extension grant types OAuth takes into account the support of additional clients or the provision of a bridge between OAuth and other trusted

Table 1
OAuth access token request example.

```
POST/token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-
urlencoded;charset = UTF-8

grant_type = authorization_code &
code = SplxlOBeZQQYbYS6WxSbIA
& redirect_uri = https%3A%2F%2Fclient%2Eexample
%2Ecom%2Fcb
```

frameworks. It also allows the definition of additional authentication mechanisms to be used by clients when interacting with the authorization server.

The OAuth 2.0 Assertion Profile [12] is an abstract extension to the specification that provides a general framework for the use of assertions as client credentials and/or authorization grants with OAuth 2.0. This profile provides a general framework for the use of assertions as client credentials and/or authorization grants with OAuth. It includes a generic mechanism for transporting assertions during interactions with a token endpoint, as wells as rules for the content and processing of those assertions. The main goal is to facilitate the use of OAuth in client–server integration scenarios where the end-user may not be present. OAuth also provides additional profile documents for standard representations of these assertions in formats such as SAML and JWT.

An assertion can be used to request an access token when a client wishes to utilize an existing trust relationship. This can be done through the semantics of the assertion, and expressed to the authorization server through an extension authorization grant type. The processes of granting authorization and assertion acquisition are outside the scope of OAuth.

Below is the list of the most important fields that should carry an OAuth Authorization Grant, along with a brief description of each:

- `client_id` (optional): the client identifier.
- `grant_type` (required): the format of the assertion defined by the AS and expressed as absolute URI.
- `assertion` (required): the assertion used as an authorization grant.
- `scope` (optional): the request may contain a scope parameter, and it is expressed as a list of space-delimited strings.

Table 2 defines the use of assertions as authorization grants. In this case, the client must include the assertion using the listed HTTP request parameters. It should be noted that the Bearer Token describes how to use bearer tokens in HTTP requests to access OAuth protected resources. Any party in possession of a bearer token can use it to get access to the associated resources.

Table 2
SAML assertion as OAuth authorization grant.

```
POST/token.oauth2 HTTP/1.1
Host: authz.example.net
Content-Type: application/x-www-form-urlencoded

grant_type = urn%3Aietf%3Aparams%3Aoauth%3Agrant-
type%3Asaml2-
bearer& assertion = PEFzc2VydGlvbiBJc3NlZUluc3RhbnQ9
IjIwMTEtMDU
[...]aG5TdGFOZWllbnQ-PC9Bc3NlcnRpb24-
```

2.3. SASL

The Simple Authentication and Security Layer (SASL) is a proposed standard (RFC 4422) [13]. The framework provides authentication and data security services in connection-oriented protocols via replaceable mechanisms, providing a structured interface between protocols and mechanisms. The authentication mechanisms can also support proxy authorization, a facility that enables one user to impersonate another.

The framework allows new protocols to reuse existing mechanisms and allows old protocols to make use of new mechanisms, describing how protocols include support for SASL, and defining the protocol for carrying a data security layer over a connection. Some examples of the currently supported SASL protocols are: IMAP, LDAP, POP, SMTP, XMPP.

Fig. 3 shows how SASL is conceptually a framework that provides an abstraction layer between protocols and mechanisms. This is possible through the use of interfaces of the abstraction layer between any protocol with any mechanism.

The IETF Working Group Common Authentication Technology Next Generation [14] is working on the definition of new SASL mechanisms for the integration of SAML [15], OpenID [16] and OAuth protocols [17]. This group defines these new SASL mechanisms in accordance with the new bridge between SASL and the GSS-API called GS2, which is defined in the RFC5801 [18]. The GSS-API is introduced in this document in Section 2.4.

The purpose of these specifications is to describe how an application client can use these security protocols over the SASL or the GSS-API to verify that an end user controls an identifier or to access a protected resource at a resource server.

2.4. GSS-API

The Generic Security Service Application Program Interface (GSS-API), defined in RFC2743 [19], provides a framework for applications to support multiple authentication mechanisms through a unified interface, but it does not actually provide security services itself. This specification provides generic fashion security services to the caller, defining services and primitives at a level independent of underlying mechanism and programming language environment, and hence allowing source-level portability of applications to different environments.

A typical GSS-API caller is itself a communications protocol, calling on GSS-API in order to protect its communications with authentication, integrity, and/or confidentiality security services. A GSS-API caller accepts tokens provided to it by its local GSS-API implementation and transfers the tokens to a peer on a remote system; that peer passes the received tokens to its local GSS-API implementation for processing. The security services available through GSS-API in this fashion are implementable (and have been implemented) over a range of underlying mechanisms based on secret-key and public-key cryptographic technologies.

As introduced at Section 2.3, a bridge is defined between SASL and the GSS-API called GS2, that allows to use GSS-API mechanisms in SASL. In this specification, both a SASL and a GSS-API mechanisms are defined and those implementors of the SASL component may also implement the GSS-API interface, working on integrating new authentication and authorization mechanisms such as OpenID and OAuth.

GSS-API provides several types of portability for applications based on the independence from mechanisms, protocols and platforms, and provides a generic interface to the mechanisms for which it has been implemented and it is independent of any communications protocol or protocol suite.

3. OAuth and WS-* integration

Today, there are more and more scenarios which are making use of OAuth to provide a simple and flexible way to authorize clients in order to access protected resources. This is due to the many advantages OAuth provides, among which we highlight the following:

- The use of REST to make requests (lightweight, easy to implement, readable results, etc.).
- The provision of a simple API which simplifies the creation of different clients. This is easier than dealing with large XMLs to request and obtain something.

Different OAuth implementations normally use different types of authorization grant and access tokens. The main complexity to reach the interoperability between OAuth implementations is to offer compatibility for these types of internal tokens.

Table 3
Mapping between OAuth access token request and WS-Trust issuance request.

```
POST/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-
urlencoded;charset = UTF-8

grant_type = urn%3Aietf%3Aparams%3Aoauth%3A
grant-type%3Asaml2-
bearer& assertion = PEFzc2VydGlvbiBJc3NlZUluc
3RhbnQ9IjIwMTEtMDU
[... omitted for brevity...]aG5TdGFOZWllbnQ-
PC9Bc3NlcnRpb24-
& scope = http%3A%2F%2Fwww.test-service.com


<RequestSecurityToken>
    <TokenType>urn:ietf:wg:oauth:2.0:oob</
TokenType>
    <RequestType>
    http://docs.oasis-open.org/ws-sx/ws-trust/
200512/Issue
    </RequestType>
    <AppliesTo>
      <EndpointReference>
       <Address>http://www.test-service.com</
Address>
      </EndpointReference>
      <AppliesTo>
  ... assertion ...
</RequestSecurityToken>
```

Table 4
WS-Trust RequestSecurityToken Response example.

```
<RequestSecurityTokenResponse>
  <TokenType>urn:ietf:wg:oauth:2.0:oob</TokenType>
  <RequestType>
  http://docs.oasis-open.org/ws-sx/ws-trust/
  200512/Issue
  </RequestType>
  <RequestedSecurityToken>
  ... access-token ...
  </RequestedSecurityToken>
  <Lifetime>
    <Created>2012–03-27T10:00:00.503Z</Created>
    <Expires>2012–03-27T10:30:00.503Z</Expires>
  </Lifetime>
</RequestSecurityTokenResponse>
```

Table 5
WS-Trust mapping to OAuth access token response example.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset = UTF-8
Cache-Control: no-store
Pragma: no-cache

{
"access_token":"... access-token",
"token_type":"urn:ietf:wg:oauth:2.0:oob",
"expires_in":1800,
"other_parameters":"other-parameters-values"
}
```

Table 6
WS-Trust token validation example.

```
<RequestSecurityToken>
  <TokenType>urn:ietf:wg:oauth:2.0:oob</TokenType>
  <RequestType>
    http://docs.oasis-open.org/ws-sx/ws-trust/
    200512/Validate
  </RequestType>
  <ValidateTarget>access_token</ValidateTarget>
</RequestSecurityToken>


  <RequestSecurityTokenResponse>
  <TokenType>
  urn:oasis:names:tc:SAML:2.0:assertion
  </ns4:TokenType>
  <Status>
    <Code>
    http://docs.oasis-open.org/ws-sx/ws-trust/
    200512/status/valid
    </ns4:Code>
    <Reason>Valid token</ns4:Reason>
  </Status>
</RequestSecurityTokenResponse>
```

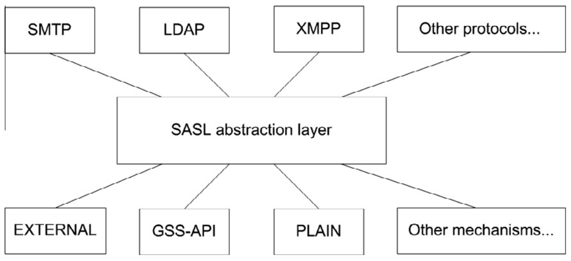

Fig. 3. SASL scheme.

The authorization grant and access tokens can be anything, for example, encrypted string, a SAML assertion [7], a X509 certificate, etc. So, the OAuth Authorization Service (AS) must be able to deal with the authorization grant types that it expects, in order to generate the access token if the authorization grant is valid.

By means of WS-* (Section 2.1), it is possible to reduce the complexity supported by the OAuth AS, so easing the interoperability through the delegation of the issuance and validation processes on the mechanisms introduced by WS-Trust (Section 2.1.2). The STS can deal with SAML, X.509 and JWT tokens (using its extension capabilities) for validation and it can use different types of policy engines to perform the access control. Furthermore it is able to generate different types of tokens such as SAML and JWT tokens and it can be extended to support more token types, such as OAuth access tokens.

On the other hand, WS-Trust arose as a standardization mechanism to offer security and interoperability between traditional Web Services (SOAP) that use heavyweight protocols such as SAML or X.509 certificates. WS-Trust is, in turn, a heavyweight protocol and does not seem to be suitable for the new scenarios which have appeared lately, such as social networks or cloud services.

However, the integration of scenarios which use OAuth with the WST STS can be very helpful to extend the capabilities for authentication and authorization, and simplify the complexity of the OAuth Authorization Server. In the same manner, there are situations where a given service using WS-* for security needs to make use of OAuth protected resource, so the integration between OAuth and WST must be done in both directions.

3.1. Scenario OAuth and the WST STS

Let us suppose a scenario in which a user tries to access a service which uses OAuth as authorization mechanism. The OAuth AS implementation only deals with a specific type of authorization grant, but the user may have a previous authorization grant, which can be of a different type from the one supported by this AS.

The question is: How can an access token be obtained to access the service using heterogeneous authorization grants?

It is in this type of situation that the WST STS can be very important. The main idea is to try to delegate the complexity supported by the OAuth AS in the Security Token Service described by WST so that the STS is responsible for authenticating, authorizing, generating and validating the tokens used.

3.1.1. Proposed solution

Fig. 4 illustrates the OAuth elements and the WST STS, and how a client, acting on behalf of a user using an authorization grant, can access a resource protected by a different OAuth implementation, that is to say, an implementation that is not able to deal with the authorization code supplied.

The operational flow of this proposed solution begins by obtaining an OAuth authorization grant based on the bearer token specification [20] (omitted in the figure). With it, the client, acting on behalf of a user, requests an access token to get access to the protected resource. The client sends the authorization grant to the Authorization Server along with other parameters (scope of the protected resource, client identifier…). At this point, the AS tries to validate the authorization grant.

Since the authorization grant type can be heterogeneous, the AS delegates to the STS the responsibility of validating the authorization grant and if it is valid, the STS will issue the requested access token. Once the access token has been obtained by the client, it will be used to access the protected resource.

The client sends the access token obtained to the Resource Server (RS). The access token must be validated by the RS so that it can return the protected resource. The RS can validate the access token by itself if it is able to deal with the type of the access token. If not, the RS can query the AS, which could either validate the access token by itself or, in the same manner as above, could delegate the process to the STS. Finally, if the access token is valid, the RS returns the protected resource.

As can be seen, it is sufficient for OAuth AS to be able to make WS-Trust requests (WS-Trust messages) to the STS and understand the WS-Trust responses. There is also the possibility that the RS can use the STS to validate the access tokens, but this would increase the complexity of the solution since the RS should be able to deal with WS-Trust messages too. These decisions imply a trust relationship between OAuth AS/RS and STS.

To enable interoperability between OAuth AS and STS, the OAuth AS must be able to translate the OAuth REST requests to WS-Trust SOAP requests, that is to say, it must be

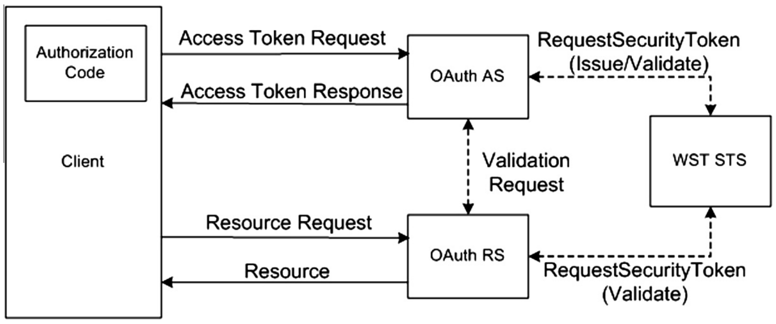

Fig. 4. OAuth-STS integration.

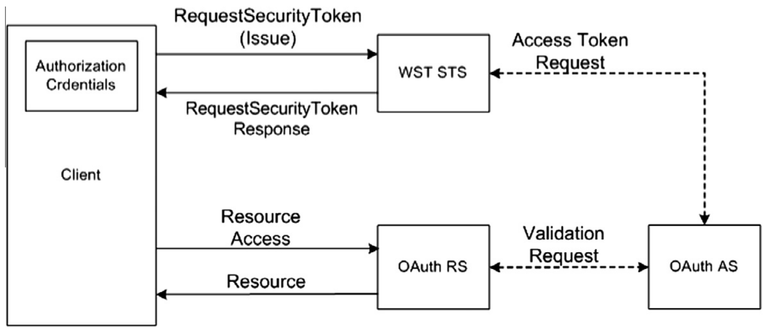

Fig. 5. STS-OAuth integration.

able to create a WS-Trust message for issuance from the REST request received.

As shown in Section 2.2 on OAuth, the Access Token Request contains the following fields: `gran_type`, `code` and `redirect_uri`. The only field which is needed to obtain the access token from the STS is the `code` (authorization grant), so the OAuth AS must add the authorization code in a WS-Trust message and send it to the STS. Table 3 shows the mapping between OAuth and WS-Trust requests to obtain an access token using a bearer token received by the AS.

The `RequestType` element specifies that the request is an issuance request and its value is fixed, in this case http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue.

According to the WS-Security specification, if the authorization grant (assertion) is a SAML assertion then it is added to the WS-Trust message as another XML element, but if it is an encoded string (such as base64 encoded string) then it is added inside a `BinarySecurityToken` element of WS-Security, indicating the value and encryption type of the code so that the STS can handle it.

If the scope parameter exists in the client request, then it is mapped as `Address` element of `EndpointReference` inside the `AppliesTo` element and, in this case, the `TokenType` element is not mandatory. This element is used to decide the type of access token to return. But if no scope is specified in the client request, then the OAuth AS must set the type of the access token it wants in the `TokenType` element of WS-Trust request. In this example `urn:ietf:wg:oauth:2.0:oob` indicates that the AS is requesting an OAuth token.

When the STS receives the request and validates the authorization grant, it responds to the OAuth AS with a WS-Trust `RequestSecurityTokenResponse` message containing either the access token if everything went well or an error indicating why the authorization grant is not valid.

Finally, the OAuth AS parses the response and extracts the access token issued by the STS and sends it back to the client in an OAuth access token response.

The following example shows the mapping between WS-Trust response (Table 4) and OAuth access token response (Table 5) to return an access token to the client to be used in an OAuth RS. That is, to translate the WS-Trust response into a JSON structure.

As can be seen, the access token returned by the STS (since it is defined by WS-Trust) is contained in the `RequestedSecurityToken` element with its token type (normally it will be the same as the requested one except when an `AppliesTo` element is used to specify the endpoint of the resource which the token will be applied to) is contained in the `TokenType` element. The `Lifetime` element states the creation and expiration dates of the token, so the AS must parse these dates in order to obtain the expiration time in seconds used by OAuth protocol. Other parameters supplied in the OAuth requests (e.g. scope) can be for the response by the AS. In this case, the OAuth access token response is a mix between WS-Trust response elements and OAuth access token request parameters.

The main advantages of this solution are the removal of part of the AS complexity and allowing different OAuth implementations to be compatible between them, so an access token obtained in an OAuth AS can be used to access protected resources from different OAuth Resource Servers.

3.2. Scenario WS-Trust STS and OAuth

In contrast, there is a scenario in which a WS-Trust client tries to access an OAuth protected resource. The client requests a valid token from the STS for the resource using WS-Trust mechanisms. It should be noted that the token issuance can be done either by the STS itself or this can request the token from a trusted OAuth AS.

The client with the OAuth token would access the resource based on the needed protocol required by the resource provider, because the resource access is out of the OAuth's scope.

3.2.1. Proposed solution

Fig. 5 shows the proposed solution for this scenario. The client, which uses WS-Trust mechanisms, has to access an OAuth protected resource. In this case, the WS-Trust client throws a `RequestSecurityToken` request to the STS indicating the needed token type and the target service.

The STS evaluates the request and returns the required token to the client using a `RequestSecurityTokenResponse` message of WS-Trust standard.

The client uses this token to obtain the protected resource from an OAuth RS. As seen in the previous scenario, this access token must be validated by the RS so that it can return the protected resource. The RS can validate the access token if it is able to deal with the type of the access token. If not, the RS can query the AS, which could either validate the access token or act as a proxy of the STS. Finally, if the access token is valid, the RS returns the protected resource to the client.

Internally, when the access token validation is delegated to the STS, the validation request made by the AS is as shown in Table 6.

The `RequestType` element specifies that the request is a validation request and its value, in this case, is fixed at http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate and the access token is the token previously obtained. The `TokenType` is not needed, so it can be omitted.

The STS responds with a WS-Trust validation response, indicating the status of the token validation. The `Code` element, contained in the `Status`, indicates the result of the validation, where the possible values are http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid and http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/invalid.

3.3. Result summary

As stated in the introduction, the proposed mapping mechanisms between WS-Trust and OAuth protocol messages (SOAP and REST messages) allows full integration between these protocols. The resulting architecture offers an easy way to extend the security features of existing solutions based on OAuth or WS-Trust.

The integrated WS-* mechanisms allow the complexity of OAuth AS to be delegated to the STS. Furthermore, this takes advantage of the interoperability benefits offered by WS-Trust and the extension capabilities offered by the STS in order to improve the OAuth AS functionality and the range of token types accepted.

The integration in both directions has been achieved, so similarly, the STS can benefit of the interoperability with the new emerged services in social network area and new web 2.0 applications.

4. Integrated design and implementation

After describing integration of OAuth and WS-* technologies, we decided to translate this to a practical (prototype) implementation and thus allow our approach to form part of a bigger solution. Therefore, we implemented the proposed approach for the GEMBus framework [21]. It is a complete framework to provide service composition in wide and federated scenarios, so it is a perfect candidate to benefit from the capabilities provided by our proposal. Also, as GEMBus is conceived for the research and education communities forming the GANT network, a European-wide research network, we find in it a large environment of already deployed and federated services that may benefit from our proposal. Furthermore, it can give us a wealth of useful scenarios to demonstrate the capabilities of our proposal.

GEMBus offers a Security Token Service (STS) which is responsible for managing security credentials. Its design is based on the guidelines given by WS-Trust standard, which focuses on SOAP services. In the case of wanting to interact with other types of services such as REST services, it should make use of a solution, like the one given in the Section 3.2.

Below we describe the GEMBus architecture, the Security Token Service, the OAuth Authorization Service, and last, the resulting integrated solution with the proposed modifications to these elements in order to achieve interoperability.

4.1. GEMBus architecture overview

GEMBus is a framework that arises inside the National Research & Educational Network (NREN) community of Europe, within the GANT project, with the aim of addressing a key paradigm in the NREN innovative service environment: seamless collaboration, moving beyond the current model of a community of collaborating network operators, into an open cloud of resource services that users can freely compose to access network resources and other applications operating on top of the network. The scenario may include both services made available by network operators and services offered by the users themselves to other users, therefore making GEMBus both a facilitator and a consumer of cross-stratum optimization mechanisms.

The self-management of networked systems and services is a key challenge for the Future of Internet (FI) [22–24]. The Future Internet Assembly (FIA) [25] released the *Management and Service-aware Networking Architectures (MANA) for Future Internet* position paper [26] to support and guide the definition of management architectures for FI and the role of management operations within network services.

Developed as part of the GANT project [27] and in collaboration with user groups to identify and prototype use cases, the GEMBus framework aims to provide a new environment to enable users to create, compose, and consume service facilities on demand. It is intended to expand the current Service Oriented Architecture (SOA) model to produce the basic framework for a federated service bus as the foundation of future advanced network services for the European research and academic community.

As stated above, the main objective of the GEMBus framework is to provide a mechanism to define, discover, access, and combine network services. Thus, the framework will be exposed to the application elements at the infrastructure-level and service-level elements. It is the basis for a federated, multi-domain service bus that will be able to integrate (compose) any network service, specially within the GANT infrastructure but open to any service-oriented infrastructure. Moreover, it allows flexible negotiation of service provision capabilities dynamically on a mutual basis, avoiding centralized service management as much as possible.

At the beginning of the research project, we studied the major use-cases for network service integration, as we described in [21,28]. From the results of this study, we determined the essential mechanisms that should be provided by the framework, with the cross-layer service orientation and composition being the most important pitfalls to overcome. Finally, we put together all requirements and defined the boundaries of the GEMBus framework by providing a set of infrastructural services to be included in the platform. Thus, to achieve its objective, the common infrastructural services included in the framework, as illustrated in Fig. 6, are the following:

- A distributed and federated registry for service descriptions to allow other services to know where to find and access services.
- A messaging service that provides a flexible services interconnection functionality with high level of dynamicity.
- An accounting service for traceability and diagnostics through logging the desired message exchanges.
- A service repository to store service components to be easily found and deployed.
- A security service to provide/enforce access control and protect/enforce the rights of users and services over other services or the whole framework.
- A service composition engine to take different component services, either deployed locally or remotely, and thus build a new composite service.

The service bus concept used in the architecture is similar to the well-known Enterprise Service Bus (ESB) concept [29]. It is used as the basis to connect the services and to address the framework objectives. A service bus offers an effective method of integrating heterogeneous services, allowing their loose coupling, so gaining flexibility and dynamism, as well as the simplification and standardization of the interfaces used by service providers and consumers.

There are two main directions in which mutual influence in the evolution of cloud infrastructures and service-oriented architectures can translate into an easier interaction with the supporting infrastructures and the user applications. First of all, service deployment and operation can greatly benefit from a supporting cloud infrastructure able to transparently provide elastically and on-demand computational and storage resources. On the other hand, cloud infrastructure services are essentially service-oriented and therefore suitable to take advantage of supporting services, like messaging, security, accounting, composition, so simplifying their integration into business processes.

In any of the above directions, multi-domain issues have to be considered from the beginning: service deployment in any cloud infrastructure beyond enterprise limits, as well as access to Cloud interfaces outside those limits require mechanisms spanning several management domains. Other, more complicated use cases like collaborating services supported by different infrastructures, or access to different cloud providers imply much more complicated settings, although they are clear application environments in the short term, if not already required.

The GEMBus framework provides the ability to compose services from the underlying virtualization systems to cover the requirements of the orchestration systems as defined in [26]. In fact, the GEMBus framework is fed by any interface that can be represented as a service in order to combine them to produce and deliver bigger services. Moreover, it can be placed in the middle of the current Cloud Computing infrastructures [30] to serve as Platform-as-a-Service (PaaS) technology and thus build PaaS services. In summary, it is a global middleware that *talks* in terms of services and produces combined functionality (composite services).

4.2. The Security Token Service

The GEMBus Security Service must provide mechanisms to ensure security, privacy and simplicity for the communication that takes place within the GEMBus archi-

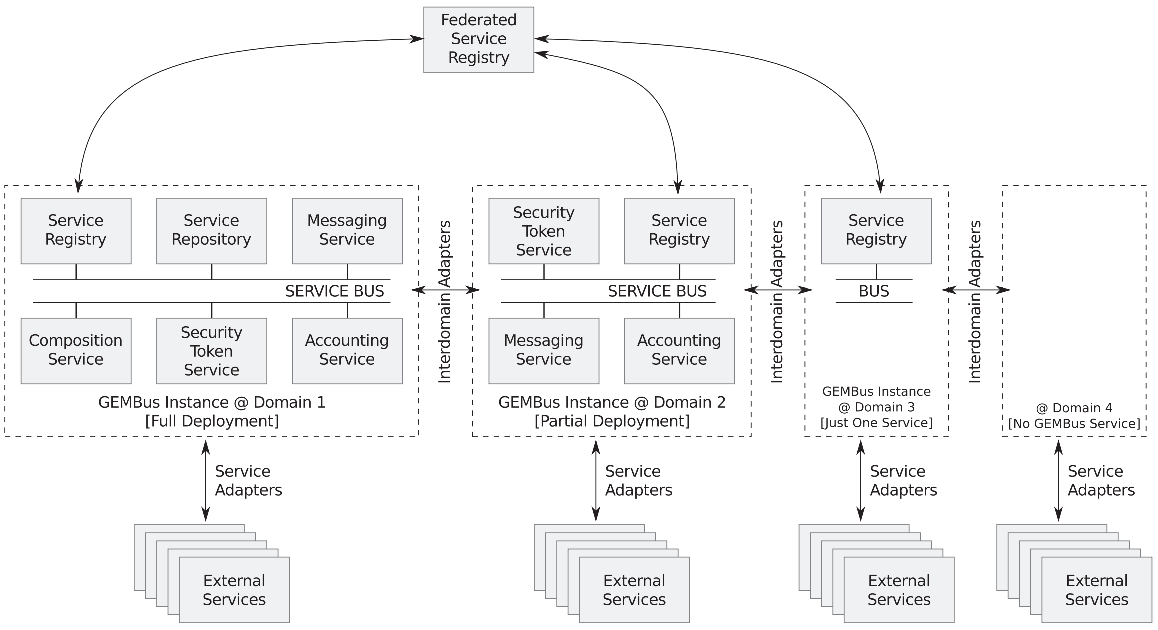

Fig. 6. Overview of the GEMBus architecture.

tecture. The Security Token Service (STS) is based on principles established by the WS-Security and WS-Trust specifications. As shown in Sections 2.1.1 and 2.1.2, the WS-Trust is a WS-* specification and OASIS standard that provides extensions to WS-Security, and it specifically deals with the issue, renewal and validation of security tokens, as well as how to establish, assess (the presence of) and broker trust relationships between participants in a secure message exchange.

4.2.1. Internal architecture

The GEMBus architecture defines its own security token service, based on WS-Trust specifications, that offers all these security features. The GEMBus concept divides STS functionality into two different elements. First, the Ticket Translation Service (TTS) is responsible for generating valid tokens in the architecture according to credentials received from the consumer. This element includes the support of current authentication methods such as SAML or X.509 certificates, as well as extension mechanisms to incorporate others in the future. Second, the token validation process is performed by the Authorisation Service (AS), which can also be associated with more complex authorization mechanisms that imply attribute request and check security policies.

Fig. 7 illustrates a scenario in which the STS, extended with support for internal session tokens, is integrated in the GEMBus architecture. In this example, the consumer obtains an identity token (a SAML assertion, for example) from an identity infrastructure. Then it sends an authentication request to the STS using the identity token. The STS validates the consumer identity token and issues a Security Token (ST) to the consumer. With the new token, the consumer sends a request message to the provider that is intercepted by an element that extracts the ST and sends a token validation request to the STS. The AS module validates the consumer token and issues a response with a validated security token, together with an optional Session Token (SeT). Finally, the interceptor passes the message to the provider, which processes the consumer request and sends a response message to the consumer.

In addition to the flow described here, the service providers (SPs) deployed in GEMBus can validate the tokens by contacting the STS.

The STS standard concept is extended in the GEMBus architecture in order to provide additional functionalities. With the aim of improving the validation process, the STS is able to request attributes from external entities such as Attribute Authorities and Identity Providers. This new information could be used alongside that of the client in order to take an authorization decision from Policy Points using eXtensible Access Control Markup Language (XACML) [31].

4.2.2. STS Ticket Translation Service (TTS)

The Ticket Translation Service (TTS) is responsible for issuing, renewing and converting security tokens, responding to consumer requests for services that require it. These operations can only be done by the TTS, unlike security token validation that can be done either by the own service or at the framework integration elements, such as interceptors, message routers or binding components.

The main TTS operations are the issuance of new security tokens based on user's identity credentials, the token renewal and the conversion of one security token type to another compatible type. Table 7 shows an Request Security Token for the issuance of a new token. The most relevant fields are: `RequestType`, `TokenType`, the `AppliesTo` and the `BinarySecurityToken` that contains the identity credentials used by the STS to authenticate the user.

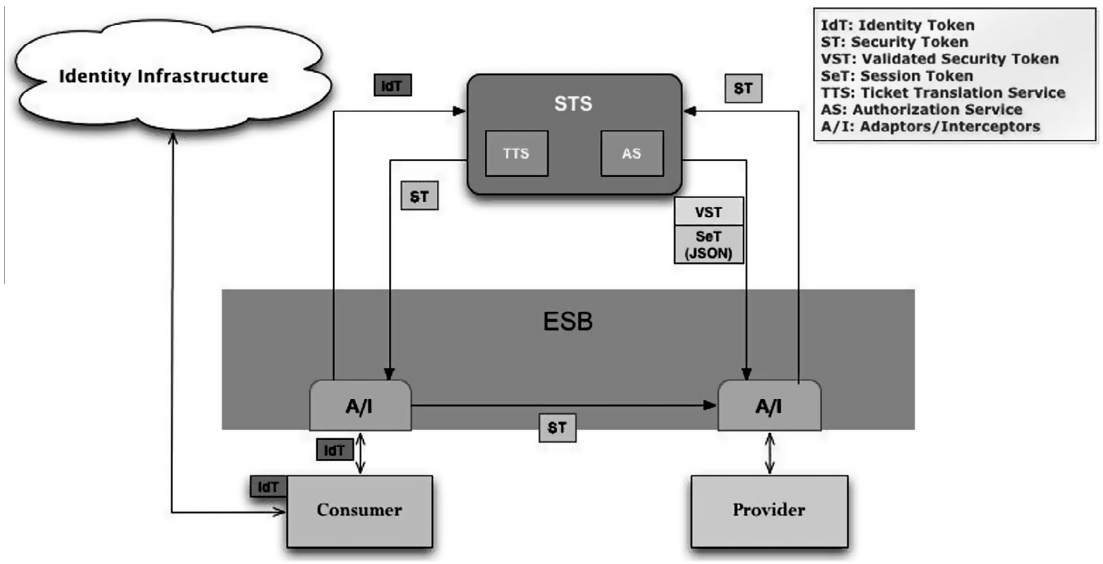

Fig. 7. GEMBus interaction scheme.

The TTS operation is as follows:

1. The consumer obtains an identity token (SAML Assertion, proof of access to a private key, etc.) from an identity infrastructure.
2. The consumer sends a request for issuance, renewal or conversion to the TTS using either the Identity Token (issuance) or a Security Token (renewal or conversion).
3. The STS validates the consumer's token (using security policies) and sends a security token to the consumer.

4.2.3. STS Authorization Service (AS)

The AS is responsible for supporting the token validation functions, responding to requests for validating tokens of consumers and services that require it.

The token validation process can be performed by the STS itself or act as a proxy, redirecting the validation process to the external service that generated it. For external validation, the Authorization Service consults an external service or IdP, and forwards the response to the STS consumer. When the Authorization Service itself performs validation, the process must verify the information contained in the token, checking the issuer, issue and expiration date, signatures, etc. In addition to the token, the Authorization Service can perform a more complex authorization process, retrieving attributes related to the token subject and consulting a Policy Decision Point (PDP) for authorization decisions.

4.2.4. Tokens

The GEMBus TTS supports transformations among different token formats, according to service descriptions as stored in the GEMBus Registry. Appropriate profile definitions describe these formats. Nevertheless, the canonical GEMBus security token (applicable by default in all GEMBus-supported exchanges) is the relayed-trust SAML assertion originally defined within the GN2 project [32] to provide identity information in scenarios where a service is acting on behalf of a user identified through an identity federation.

The WS-Security specification allows a variety of signature formats, encryptions algorithms and multiple trust domains. It is open to various security token models, such as X.509 certificates, userid/password pairs, SAML assertions and custom-defined tokens, as seen in Section 2.1.1.

JSON Web Token (JWT) [33] defines a compact token format intended for space constrained environments such as HTTP Authorization headers and URI query parameters that can encode claims transferred between two parties. The claims in a JWT are encoded as a JSON object [34] which is then optionally digitally signed and transmitted base64url encoded. JSON is a lightweight, text-based, language-independent data interchange format that defines a small set of formatting rules for the portable representation of structured data.

WS-Security does not define a specific profile for this kind of tokens. In order to make the use of JWT possible, the GEMBus STS uses the extension capabilities of the WS-Security based on the Binary security tokens element. It is used to embed X.509 certificates and Kerberos tickets, as well as other non-XML formats that require a special encoding format for inclusion. The `<wsse:BinarySecurityToken>` element defines two attributes that are used to interpret it. The ValueType attribute indicates what the security token is and the EncodingType that tells how the security token is en-

Table 7
Request Security Token (issuance) example.

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:sts="http://sts.wstrust.security.gembus.geant.net/">
  <soapenv:Header/>
  <soapenv:Body>
    <ns4:RequestSecurityToken
    xmlns="http://www.w3.org/2005/08/addressing"
    xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:ns3="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:ns4="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
    xmlns:ns5="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <ns4:RequestType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue</ns4:RequestType>
      <ns4:TokenType>urn:geant:gembus:security:token:1.1:gemtoken</ns4:TokenType>
      <ns2:AppliesTo>
      <EndpointReference>
          <Address>www.sp2-gn3.org</Address>
        </EndpointReference>
      </ns2:AppliesTo>
      <ns4:Lifetime>
        <ns3:Created>2012–02-20T12:00:00.574Z</ns3:Created>
        <ns3:Expires>2012–02-20T14:30:00.574Z</ns3:Expires>
      </ns4:Lifetime>
      <ns5:BinarySecurityToken ValueType="X509v3" EncodingType="Base64Binary">
      MIIC...b21lLVNOYXRlMRAwDgYDVQFJssZO=
     </ns5:BinarySecurityToken>
    </ns4:RequestSecurityToken>
  </soapenv:Body>
</soapenv:Envelope>
```

Table 8
Internal GEMBus token example.

```
<wsse:BinarySecurityToken
  EncodingType="Base64Binary"
  ValueType=
    "urn:geant:gembus:security:token:1.1:gemtoken">
    ...
</wsse:BinarySecurityToken>
```

coded. An example of the internal GEMBus Token is shown in Table 8.

4.3. OAuth Authorization Service

The OAuth Authorization Service is the entity in charge of authenticating the credentials of users and clients, validating the authorization grants and generating the access tokens. The AS selected for this demonstration was developed by RedIRIS Institution as a testbed to allow the development of prototypes and interoperability models for future use.

Its core makes use of an OAuth library implemented following the 14th draft version of the standard and offers access to its functionality through a REST interface.

4.4. Integrated architecture

The proposed architecture that allows the integration between WS-* and OAuth systems is composed of four main roles: the client of the system that needs a specific token, the WS-Trust STS and the OAuth AS that are responsible for providing the requested token, and finally the service which the user wants to access and that will consume the token. Fig. 8 shows the relationships between these roles.

The architecture proposed by GEMBus is based on message exchanges performed by different services that can be connected in many ways. Since the ESB is the main integration mechanism provided by GEMBus, and it can also act as a container, it is possible to develop and deploy a service directly on the bus. GEMBus offers, furthermore, integration capabilities, such as interceptors, message routers and binding components. Whether deployed inside the bus or running as an external service, the STS can be used directly through a SOAP interface.

The GEMBus framework has been developed over FUSE ESB, adding to it the complete GEMBus ecosystem of services to enrich the original FUSE platform functionality. FUSE ESB is an open source integration platform with a pluggable architecture oriented to SOA solutions. The GEMBus modifications are focused in the STS, which has been extended with a module and now allowing the translation to REST OAuth message in order to use an OAuth AS for issuing tokens.

The OAuth Authorization Service development is based on the Draft 14 of OAuth specification. This element of the architecture is accessible through a REST interface which plays the role of the facade of the security system, but it has been adapted to support SOAP communications and to understand WS-Trust messages to interact with the STS.

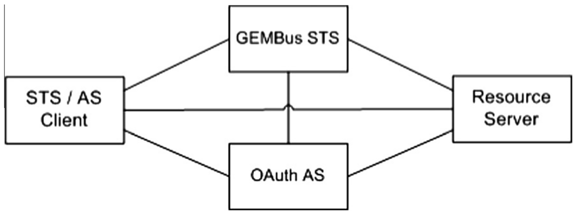

Fig. 8. Integration relationship diagram.

As an additional feature, the extensions made to both STS and OAuth AS would allow us to see the OAuth AS like the REST interface of the STS and viceversa, the STS like the SOAP interface of the OAuth AS, that is to say, each security service may use the interface of the other service to manage requests.

Table 9 shows a list of main software components that take part in the testbed. The table includes software versions and the role that they play. On the other hand, the Table 10 shows the most relevant standards used and implemented in the testbed demo. The software used in the demo has been developed from the beginning with the aim of making it flexible and configurable through configuration files. This allows us to use the same software in both scenarios only by adapting the settings.

5. Deployment and evaluation

As part of this work, we set up a functional prototype where the concepts described in this paper are put into practice. Using this, the objective of this section is to evaluate the proposed architecture in two different scenarios, one for the OAuth accessing the STS and other for the STS accessing OAuth. Thus, we aim to demonstrate the feasibility and validity of integration models.

To perform the evaluation, we deployed the functional prototype in a testbed composed of a single machine with the GEMBus software (including the STS), the OAuth Authorisation Server, and the OAuth Resource Server. We group these elements into one single machine to focus the evaluation on the software part and not on the network. In practice, we do not consider the cost of network hops because it is almost the same in both scenarios. Moreover, the equipment used to obtain the measurements is an Intel (R) Pentium (R) Core 2 Duo CPU 2400 MHz, with 8 GB of RAM, running Mac OS X Lion 10.7.4. It has enough resources to host all components without disturbing each other.

Using this testbed and components we performed 500 executions of the whole process (the token request for

Table 9
Used software.

Element	Software	Version
ESB	FUSE ESB	4.3.1
SOA framework	GANT GEMBus	1.0.0
WS-* implementation	GEMBus STS	1.0.0
OAuth library	RedIRIS OAuth library	1.0.0
OAuth AS	RedIRIS OAuth AS	1.0.0

Table 10
The used standards and their versions.

Standard	Version
WS-Security	1.1
WS-Trust	1.4
SAML	2.0
OAuth	Draft 14

both scenarios), to obtain the time measurements of the different steps of each process. Therefore, each scenario has been decomposed into several steps in order to analyse the time employed by each flow element, as can be seen in detail in the following sections. First, we start with the execution of a standalone scenario where the GEMBus STS and OAuth solutions run separately without any integration. Then, we proceed with the execution of the proposed integrated architecture.

5.1. Scenario 0: standalone execution

In this section we analyse standalone scenarios, where the STS and OAuth are running without being integrated, to obtain the base case to compare the scenarios using the integrated solution proposed in this paper. From them, we measured the time spent in the reception and initial processing of the client requests, the time spent by the OAuth Authorization Server (AS) and GEMBus Security Token Service (STS) in the processing, and the time spent sending of the results back to the client. With the results from this evaluation we can perform an objective analysis of the proposed approach.

Fig. 9 shows the resulting times measured for the standalone scenarios where each security service (the OAuth AS or the GEMBus STS) produces its default token. As we can see in the figure, the times are relatively similar. The biggest difference is seen at the second step, where the specific security service has to check the validity of the credentials and then generate the new token. The main reason for this difference is that while the STS checks the signature of the credentials, the AS only checks some dates and fields of the request, because there is a previous trust relationship between the Client and the AS.

5.2. Scenario 1: OAuth AS accessing WST STS

In this section we analyse the scenario proposed in Section 3.2 in which an OAuth Client wants access to an OAuth protected resource that does not use the standard authorization token. For this reason, the OAuth Authorization Server (AS) delegates the generation of the token to the Security Token Service (STS). The interaction between the elements of this integration proposal are shown in Fig. 4. The detailed steps of the scenario are as follows:

1. The OAuth Client sends an OAuth Access Token request using the REST interface of the OAuth AS including the required credentials, in this case a SAML 2.0 assertion. This step includes the message reception by the OAuth AS.
2. The OAuth AS validates the credential, and creates and sends a WS-Trust Request Security Token request to the GEMBus STS including the required credentials.

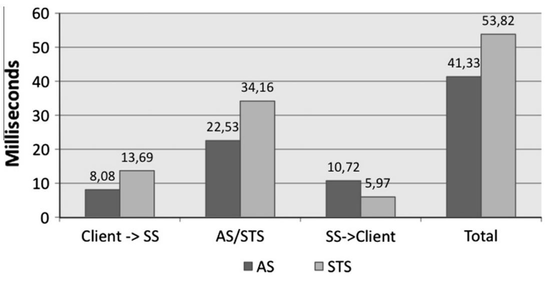

Fig. 9. Means of execution times for standalone scenarios.

3. The GEMBus STS receives the request through its SOAP interface, validates the client credentials (including the signature) and generates a GEMBus Token in JWT format.
4. The GEMBus STS returns to OAuth AS a WS-Trust Request Security Token response that the AS has to process (convert) to obtain an OAuth Access Token response.
5. Finally, the AS returns the response to the Client, and the latter receives it for subsequent operations.

To obtain the time measurements in the integrated scenarios, we modified the components to log the time spent in the execution of each step. After the execution of the tests, we analysed the log file to get their statistical data. Specifically, we calculated the mean of the time required to perform each step, the standard deviation, the confidence Interval to 95%, and the percent of the total time. These values are shown in Fig. 10 and Table 11.

To summarize, the time measures obtained show that the heaviest step of the flow is the validation of the credentials and the token generation carried out by the STS in the third step ("STS: generation"). Thus, the STS needs 53% of the total time to complete these tasks. In the second place, the AS consumes 23% of the total time in processing the client request and asking the STS in the second step ("AS: validation"), and converting the STS response into the OAuth response. The remaining time is spent on the interactions with the Client.

5.3. Scenario 2: WST STS accessing to OAuth AS

We have just studied the scenario where an OAuth client accesses an OAuth protected resource, in this section we proceed to study the scenario where a WS-Trust client accesses an OAuth protected resource. This scenario is discussed in Section 3.2 and Fig. 5 shows the interactions between the elements of the integrated architecture. It comprises the following steps:

1. The Client sends a WS-Trust Request Security Token request over SOAP to the GEMBus STS, including the required credentials, in this case a SAML 2.0 assertion. This step also includes the message reception by the STS and its unmarshalled process to discover the type of required token.

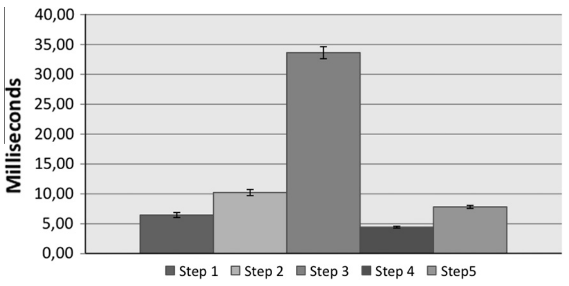

Fig. 10. Scenario 1 – Means of execution times for the different steps (operations). Step 1 is the "Client → AS", step 2 is the "AS: validation", step 3 is the "STS: generation", and step 4 is the "AS → Client".

Table 11
Scenario 1 – statistical data obtained from the measured times (in ms).

Step	Mean	Std. dev.	C.I. 95%	Time %
Client → AS	6.46	4.79	0.42	10.32
AS: validation	10.23	5.76	0.50	16.35
STS: generation	33.63	11.48	1.01	53.73
AS: issuance	4.44	1.80	0.16	7.09
AS → Client	7.83	2.84	0.25	12.51
Total time	62.58	18.64	1.63	100.00

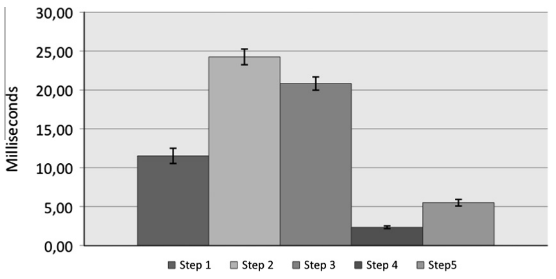

Fig. 11. Scenario 2 – Means of execution times for the different steps (operations). Step 1 is the "Client → STS", step 2 is the "STS: validation", step 3 is the "AS: generation", and step 4 is the "STS → Client".

Table 12
Scenario 2 – statistical data obtained from the measured times (in ms).

Step	Mean	Std. dev.	C.I. 95%	Time %
Client → STS	11.52	11.17	0.98	17.88
STS: validation	24.25	11.46	1.00	37.64
AS: generation	20.83	9.72	0.85	32.32
STS: issuance	2.34	2.01	0.18	3.63
STS → Client	5.50	4.81	0.42	8.53
Total time	64.44	26.89	2.36	100.00

2. The GEMBus STS validates the credentials (including the signature) and creates an OAuth Access Token Request for the OAuth AS.
3. The OAuth AS receives the request through its REST interface, validates client credentials and generates an OAuth Access Token in JWT format.
4. The AS returns an OAuth Access Token response to the GEMBus STS. It is processed by the STS and converted into a WS-Trust Request Security Token response.
5. Finally, the STS returns the response to the Client, and the latter receives and keeps it for subsequent operations.

As we did in the previous section, to obtain the time measurements in the integrated scenarios we modified the components to log the time spent on the execution of each step. Then, we analysed the log file of the application to extract statistical data from the measurements. Again we specifically calculated the mean of the time required to perform each step, its standard deviation, the confidence Interval to 95%, and the percent of the total time. These values are shown in Fig. 11 and Table 12.

For this scenario, the time measures obtained show that the heaviest step of the flow is the validation of the credentials and the token generation carried out by the STS in the second step ("STS: validation"). Thus, the STS needs 37.64% of the total time to complete these tasks. In the second place, the AS consumes 32.32% of the total time when generating the OAuth Access Token ("AS: generation"). The fourth step ("STS: issuance") takes only 3.63% of the total time. Finally, the remaining time is spent on the interactions with the Client.

5.4. Discussion

In this section we discuss the results obtained in the evaluations performed in the two different scenarios and compare them with the results obtained in the standalone scenarios.

We discussed above that the steps that take a long time are, for all scenarios, those that validate and generate tokens. Comparing these times with the standalone results, we can effectively see that they are very similar. The generation step of the first scenario takes 33.63 ms and the STS in the standalone scenario takes 34.16 ms. For the OAuth part, the second scenario takes 20.83 ms and the standalone scenario takes 22.53 ms. For the total time, we can see that, on the one hand, that the first scenario takes 62.58 ms while the standalone OAuth solution takes 41.33 ms. This little overhead is clearly introduced by the extra STS generation step, so the cost of the added functionality is feasible. On the other hand, the second scenario takes 64.44 ms while the standalone STS solution takes 53.82 ms. Here, the difference is smaller and is only half the time spent by the OAuth AS generation step. This demonstrates a small overhead for the gains obtained from the integrated solution.

When comparing the two integrated scenarios with each other, from the total time measurements in both scenarios we can see that they are very similar. However, the time distribution is more homogeneous among the processing steps of the second scenario. The STS in the second scenario consumes more time than the OAuth AS, mainly due to the credential signature verification. Even so, the AS gains more prominence in the second scenario due to the validation and generation of new tokens. The first step consumes more time in the second scenario than the first because WS-Trust messages are more complex than OAuth messages.

In summary, as we mentioned, the total times of the integrated scenarios are slightly higher than the standalone scenarios. However, relating the results to overhead percentage, the integrated solution takes a 56% more time than the standalone case for the OAuth operations and 16% more time than the standalone case for the STS operations.

Although this overhead may seem large, in terms of absolute cost, taking into account that these operations are only run during authentication, the overhead is affordable and thus we demonstrate the feasibility of the proposed solution. Anyway, the time overhead could be reduced with the optimization of the code but, even so, the benefits of the proposed integration of these two protocols for improving federation and interoperability are, in our opinion, much greater.

6. Conclusions and future work

This paper presents the current scenarios for authorization found in the context of web services and new platforms such as social networks. There are a variety of solutions to integrate technologies, but new security protocols arise faster than the integration mechanisms. New scenarios need lightweight protocols that ease the development of new services, as well as the integration among them, to allow resource sharing. As discussed throughout the paper, OAuth is becoming the *de facto* standard for such authorization task.

That said, the paper also presents a route through different integration mechanisms for security protocols, focusing analysis on the WS-* specification family. Moreover, the GEMBus project has chosen this family of specifications to give security mechanisms to its framework, since it is specially oriented to the interoperability of web services. At this point, it is notable that both WS-Security along with WS-Trust and OAuth offer powerful extension mechanisms to be adapted to new security protocols.

After introducing the authorization and general security technologies used within Web services mechanisms, the paper presents two different scenarios to set out the integration from different points of view. These scenarios outline the interoperability problem between services and resource providers that implement different security protocols. As a possible solution, the paper proposed the necessary integration and translation mechanisms to allow the interconnection of heterogeneous services from both the viewpoint of a WS-Trust service which wants to consume OAuth resources as well as when an OAuth architecture wants to integrate with a WS-Trust architecture to take advantage of the Security Token Service (STS).

On the one hand, the use of these integration mechanisms allows OAuth architectures to reduce the complexity of their authorization server by delegating part of their functions to the STS. On the other hand, the extension of WS-Security to give support to OAuth tokens and the mapping made between OAuth and WS-Trust messages allows OAuth resources to be used.

We evaluate the proposed architecture in two different scenarios to prove its feasibility and viability, thus obtaining a performance profile of the execution process. We demonstrate that, in terms of absolute cost, the overhead introduced by the integrated solution is affordable and thus we demonstrate the feasibility of the proposal. However, future optimizations should be taken into account in order to reduce the overhead when instantiating the solution in production environments.

References

[1] K. Lawrence, C. Kaler, A. Nadalin, Web Services Security: SOAP Message Security 1.1 (WS-Security 2004), OASIS Standard, 2006 <http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-errata-os-SOAPMessageSecurity.pdf>.

[2] OASIS (Organization for the Advancement of Structured Information Standards), 2012 <http://www.oasis-open.org>.

[3] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), W3C Recommendation, 2007 <http://www.w3.org/TR/soap12-part1/>.

[4] N. Ragouzis, Security Assertion Markup Language (SAML) V2.0 Technical Overview, OASIS Committee Draft, 2008.

[5] C. Neuman, S. Hartman, K. Raeburn, The Kerberos Network Authentication Service (V5) – RFC 4120, Network Working Group – Internet Engineering Task Force (IETF), 2005 <http://tools.ietf.org/html/rfc4120>.

[6] D. Cooper, S. Santesson, S. Farrell, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile – RFC 5280, Network Working Group – Internet Engineering Task Force (IETF), 2008 <http://tools.ietf.org/rfc/rfc5280.txt>.

[7] S. Cantor, Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0 (SAML Core), OASIS Standard, 2005.

[8] ITU Telecommunication Standardization Sector, 1993 <http://www.itu.int/ITU-T/> (last access date 10.03.2012).

[9] Public-Key Infrastructure (X.509), 2010 <http://www.ietf.org/html.charters/pkix-charter.html>.

[10] K. Lawrence, C. Kaler, Web Services Security: X.509 Certificate Token Profile 1.1, Web Services Security (WSS), 2006 <http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-errata-os-x509TokenProfile.pdf>.

[11] A. Nadalin, WS-Trust 1.4, OASIS Standard, 2009 <http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/os/ws-trust-1.4-spec-os.pdf>.

[12] C. Mortimore, M. Jones, B. Campbell, Y. Goland, OAuth 2.0 Assertion Profile (draft-ietf-oauth-assertions-01), 2011 <http://tools.ietf.org/html/draft-ietf-oauth-assertions> (last access date 15.03.2012).

[13] A.M., K.Z. (Ed), Simple Authentication and Security Layer (SASL) – RFC 4422, Network Working Group – Internet Engineering Task Force (IETF), 2006 <http://tools.ietf.org/html/rfc4422>.

[14] IETF WG Common Authentication Technology Next Generation (kitten), 2012 <http://datatracker.ietf.org/wg/kitten/>.

[15] K. Wierenga, E. Lear, S. Josefsson, A sasl and gss-api mechanism for saml (draft-ietf-kitten-sasl-saml-09), 2012 <http://datatracker.ietf.org/doc/draft-ietf-kitten-sasl-saml/> (last access date 15.03.2012).

[16] E. Lear, H. Tschofenig, H. Mauldin, S. Josefsson, A sasl and gss-api mechanism for openid (draft-ietf-kitten-sasl-openid-08), 2012 <http://datatracker.ietf.org/doc/draft-ietf-kitten-sasl-openid/> (last access date 15.03.2012).

[17] W. Mills, T. Showalter, H. Tschofenig, A sasl and gss-api mechanism for oauth (draft-ietf-kitten-sasl-oauth-00), 2011 <http://datatracker.ietf.org/doc/draft-ietf-kitten-sasl-oauth> (last access date: 15.03.2012).

[18] S. Josefsson, N. Williams, Using Generic Security Service Application Program Interface (GSS-API) Mechanisms in Simple Authentication and Security Layer (SASL): The GS2 Mechanism Family, Internet Engineering Task Force (IETF), 2010 <http://tools.ietf.org/html/rfc5801>.

[19] J. Lian, Generic Security Service Application Program Interface Version 2, Update 1, Network Working Group – Internet Engineering Task Force (IETF), 2000 <http://tools.ietf.org/pdf/rfc2743>.

[20] M. Jones, D. Hardt, D. Recordon, The OAuth 2.0 Authorization Protocol: Bearer Tokens (draft-ietf-oauth-v2-bearer-15), 2011 <http://tools.ietf.org/html/draft-ietf-oauth-v2-bearer-15>.

[21] P. Martinez-Julia, D.R. Lopez, A.F. Gomez-Skarmeta, The GEMBus framework and its autonomic computing services, in: Proceedings of the International Symposium on Applications and the Internet Workshops, IEEE Computer Society, Washington, DC, USA, 2010, pp. 285–288.

[22] E. Al-Shaer, A. Greenberg, C. Kalmanek, D.A. Maltz, T.S.E. Ng, G.G. Xie, New frontiers in internet network management, ACM SIGCOMM Computer Communication Review 39 (2009) 37–39.

[23] J. Schonwalder, M. Fouquet, G. Rodosek, I. Hochstatter, Future internet = content + services + management, IEEE Communications Magazine 47 (2009) 27–33.

[24] A. Pras, J. Schonwalder, M. Burgess, O. Festor, G. Perez, R. Stadler, B. Stiller, Key research challenges in network management, IEEE Communications Magazine 45 (2007) 104–110.

[25] European Future Internet Portal, Future Internet Assembly, 2010 <http://www.future-internet.eu/home/future-internet-assembly.html>.

[26] A. Galis, et al., Position Paper on Management and Service-aware Networking Architectures (MANA) for Future Internet, 2010 <http://www.future-internet.eu/fileadmin/documents/prague_documents/MANA_PositionPaper-Final.pdf>.

[27] DANTE Ltd., GÉANT Project, 2010 <http://www.geant.net>.

[28] P. Martinez-Julia, A. Marin Cerezuela, A.F. Gomez-Skarmeta, A service oriented architecture for basic autonomic network management, in: Proceedings of the IEEE Symposium on Computers and Communications, IEEE Computer Society, Washington, DC, USA, 2010, pp. 805–807.

[29] D. Chappell, Enterprise Service Bus, O'Reilly Media, Inc., 2004.

[30] G. Lawton, Developing software online with platform-as-a-service technology, Computer 41 (2008) 13–15.

[31] T. Moses, eXtensible Access Control Markup Language (XACML) Version 2.0, OASIS Standard, 2005 <http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf>.

[32] DANTE Ltd., et al., GÉANT 2 Project, 2001 <http://www.geant2.net/>.

[33] M. Jones, D. Balfanz, J. Bradley, Y. Goland, J. Panzer, N. Sakimura, P. Tarjan, Json web token (jwt), 2011 <http://tools.ietf.org/html/draft-jones-json-web-token> (last access date: 15.03.2012).

[34] D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON) – RFC4627, Network Working Group, Internet Engineering Task Force (IETF), 2006 <http://tools.ietf.org/html/draft-jones-json-web-token> (last access date: 15.03.2012).

Elena Torroglosa-García is a researcher in the Department of Information and Communications Engineering of the University of Murcia since 2008 where she received her B.S. degree in Computer Science. She participated in Secure Widespread Identities for Federated Telecommunications (SWIFT) as part of the 7th Framework Program and currently is involved in the GEMBus (JRA3-T3), as part of the GN3 project (FP7-2007-2013). Her research interests include network security and identity management.

Antonio D. Pérez-Morales received the degree in Computer Engineering from the University of Seville in 2009 and he is finishing a Master in Engineering and Technology of Software. Since 2008 he is working in RedIRIS, the national research and educational network from Spain. He has participated in several European projects like pkIRISGrid and SCS and he has also participated (among others) in the GEMBus task (JRA3-T3) which is part of the GN3 project. His main interests are digital identity, security and everything related with web and mobile applications.

Pedro Martinez-Julia received the B.S. degree in Computer Science from the Open University of Catalonia in 2009 and the M.S. degree in Advanced Information Technology and Telematics from the University of Murcia in 2010. Since 2009 he is a research fellow and Ph.D. candidate in the Department of Communication and Information Engineering at the University of Murcia. He is the task leader of GEMBus (JRA3-T3), as part of the GN3 project (FP7-2007-2013). His main interests are the overlay networks, security, and distributed systems. He is reviewer of many international journals and an active associate member of ACM and IEEE.

Diego Lopez received his MS from the University of Granada in 1985, and his Ph.D. degree from the University of Seville in 2001. He joined Telefonica I+D in 2011 as a Technology Expert on network middleware and services. He has previously worked for several private and public organizations, developing and deploying communication services, most notably as responsible for Middleware Activities in RedIRIS (the Spanish National Research and Educational Network). He is one of the European liaisons to Internet2 MACE (Middleware Architecture Committee for Education), and was appointed as a member of the EC's High Level Expert Group on Scientific Data e-Infrastructures. He is one of the main architects of eduGAIN, the Authentication and Authorisation Infrastructure (AAI) of GÉANT and has contributed to the e-IRG whitepapers. His current interests are related to network intelligence and virtualization, infrastructural services, and new network architectures.