High-Performance Computing on the Web: Extending UNICORE with RESTful Interfaces

Bernd Schuller, Jedrzej Rybicki
Jülich Supercomputing Centre
Forschungszentrum Jülich GmbH
Jülich, Germany
Email: {b.schuller,j.rybicki}@fz-juelich.de

Krzysztof Benedyczak
Interdisciplinary Center for
Mathematical and Computational Modelling
Warsaw, Poland
Email: golbi@icm.edu.pl

***Abstract*—UNICORE (UNiform Interface to COmputing REsources) is a Grid middleware for accessing high-performance computing capabilities and storage resources in a secure and seamless fashion. In its current version (7.0), it offers web services using SOAP (Simple Object Access Protocol) in conjunction with a security stack based on the Security Assertions Markup Language (SAML) and the WS-Security specification. To accommodate recent integration use cases, the need for more lightweight ways to access resources through UNICORE has arisen. This work describes the architecture, design, and first implementation results of an interface to UNICORE services based on the REST (Representational State Transfer) architectural style. Crucial boundary conditions included a lightweight security layer, and full interoperability with the existing SOAP-based interfaces. This RESTful interface will greatly simplify access to and interaction with the UNICORE services and enable new use cases. It will allow integrating high-performance computing and data management services into web-based and mobile applications.**

Keywords–UNICORE; REST; Security; High-Performance Computing

I. Introduction

UNICORE was developed in the course of several German and European projects since 1997 [1]. It is a mature software suite for building federated systems and Grids. It is deployed and used in a variety of settings, from small projects to large (multi-site) infrastructures involving high-performance computing (HPC) resources. UNICORE can be characterized as a vertically integrated Grid system, that comprises the full software stack from clients to various server components down to the components for accessing the actual compute or data resources. Its basic principles are abstraction of resource-specific details, openness, interoperability, operating system independence, security, and autonomy of resource providers. In addition, the software is easy to install, configure and administrate. UNICORE software is available as open source from the SourceForge repository [2] under a permissive, commercially friendly license.

The UNICORE services can be accessed through a SOAP web service stack, realising stateful services through the Web Service Resource Framework (WSRF) specification [3]. The security layer is based on Transport Layer Security (TLS), SAML, and XML digital signatures. All these are open, well-documented standards, and in principle it is possible to implement clients to access the services in any language, and Web Service tooling exists for many programming languages. However, practical experience has shown that due to the high complexity of WSRF and SAML only Java and C# have been used. In fact, UNICORE had to provide an implementation of the WSRF specification, since none of the Java web service toolkits offers one.

Consequently, it can be difficult or even impossible to use the current Web Service APIs offered by UNICORE. For example, this may occur when integrating UNICORE services into existing applications or community workflows using different technologies than the above mentioned Java and C#. Thus, simpler, more easily accessible APIs are required. To this end we are working on two concrete use cases. The first one originates from the European Human Brain Project [4]. UNICORE will form the basis for the project's HPC Platform. It comprises of four major HPC sites, cloud storage and other resources. Here, developers want to write Python applications for accessing services of the HPC Platform, such as job management or data transfer. Lightweight mechanisms, such as OpenID Connect (OIDC) [5], should be used for authentication. The second use case is a standalone client for the UNICORE file transfer protocol (UFTP) [6], which allows users to access their data without requiring the use of a full UNICORE client. Common for the both use cases is the requirement for strong authentication and delegation of rights. Subsequently, it is important to stay compatible with the usual access through UNICORE despite the introduction of the new interfaces.

A popular alternative to SOAP are RESTful services [7]. These are usually more lightweight and more easily accessible for a number of reasons. Typically, they make use of JSON [8] instead of XML for resource representations and exploit HTTP semantics for the resource manipulation instead of defining their own. To make implementation of clients easier, one would like to avoid having to handle digital signatures on the client, so more lightweight mechanisms, such as OpenID-Connect, are of interest.

The remainder of the paper is organized as follows. Section II describes the UNICORE services and service container and how RESTful services have been realized within this context. A review of the UNICORE security solution as it applies to this work is given in Section III. The initial APIs and some first performance results are given in Section IV. The paper concludes with an outlook and the next steps.

II. Services, interfaces and technologies

UNICORE is a four-tiered system, consisting of the client, gateway, services and target system tiers. All components with the exception of the target system tier are implemented in Java.

The *Gateway* is essentially a HTTPS reverse proxy, that serves as a firewall transversal point to avoid having to configure many open firewall ports. The Gateway forwards information about the connecting client (such as the IP address or the client's SSL certificates) to the servers behind it.

The *UNICORE/X* server is the central component of a UNICORE installation. It is built around the XNJS execution engine [9], which provides the execution backend and communicates with the target system tier, and a set of service interfaces. The basic services include a Registry service that provides information about available services, job submission and management services as well as file access and file transfer.

Finally, the target system tier consists of the interface to the local operating system, file system and resource management (batch) system. This *target system interface* (TSI) is responsible for submitting jobs, performing file I/O, and checking job status. The TSI is implemented in Perl, as a server running on the resource (e. g., the login node in case of a compute cluster).

The WSRF specification introduces the concept of service instances, which can be individually addressed, and are comparable to objects in an object-oriented system. The conceptually most important UNICORE services are listed in the following, where many instances of each service will usually exist in a UNICORE container:

- *Sites* are abstracted compute resources. They have a set of properties (e. g., number of cores), have a set of storages attached, and accept job submissions.
- *Storages* are abstracted file system like data resources, which offer typical operations, such as listing files. To give access to files, storages act as factory services for file transfer resources.
- *Jobs* represent actual compute jobs on the underlying batch system. Jobs always have a working directory, which is accessible through a Storage resource. To create a new job, a job description and optionally some input data is required. The job description details what is to be executed, gives the required resources (e. g. number of CPUs), and a list of date files to be staged in and result files to be staged out. Jobs are submitted to a Site resource.
- *File transfers* are used to read or write to a physical remote file. UNICORE supports both client-server data transfers and server-server transfers, with several available data transport protocols.
- *Site factories* support virtualization technologies, since a Site is always created through a Site factory.
- *Storage factories* allow the creation of storage service instances, and can support multiple backends (e. g. plain file systems or the Hadoop file system).

The services are hosted in a container called UNICORE Services Environment (USE) that is built from well-established open source components, such as Apache CXF, Jetty and many others. USE provides the web server, security layer, service configurations, a persistence subsystem and the base classes on which the actual services are built.

Care has been taken to decouple the front-end service implementations (e. g. SOAP WSRF) from the internal state and from the representations that are sent to the clients, aiming to implement the model-view-controller pattern.

Building upon Apache CXF, RESTful services following the JAX-RS standard [10] can be deployed in the USE. The RESTful services can access all USE subsystems (e. g. for persistence) and can thus access the same resources as the WSRF services. One positive side-effect of this approach is that the same state (e. g. storage) can be accessed consistently through different interfaces.

III. Security

The flexible security system in UNICORE is one of its main assets. In this section, we briefly describe how *authentication*, *authorization* and *delegation of rights* work in UNICORE.

The goal of the *authentication* process is that the UNICORE/X server knows the X.500 name (distinguished name, DN) of the user and has verified that it is correct. Traditionally, authentication required that each entity in the system (users and servers) required an X.509 end-entity certificate. This was used for establishing SSL connections where both parties (client and server) could check that the other party was trusted. In UNICORE 7, the security architecture has been made much more flexible through the new Unity service [11], which can authenticate users using some other means (e. g., username and password). Client certificates are no longer necessary, though they can still be used. Server certificates are still required, for instance for securing the communication channel.

The *authorization* process takes the DN established by authentication and maps it to a set of user attributes, which are used for two purposes. First, the server's access control policies (written in XACML) are evaluated to decide whether the user is allowed to perform the current operation. Second, the attributes are used later by the services' business logic. For example, two typical attributes are the local Unix username and groups, which are required, e. g., for job submission.

All of the authentication and authorization process is configurable by the UNICORE administrator, in accordance with the principle of autonomy of the resource provider.

Finally, *delegation* is needed, for example when a user submits a job that requires data to be downloaded from another UNICORE server. In such a case it is not acceptable to impersonate the user by passing along her credentials. This would pose a potential security risk. Instead, a delegation token is required that cryptographically asserts the original user's identity and asserts that the original user delegates rights to the server (for performing particular action on her behalf). Delegation is implemented in UNICORE via signed SAML assertions, where the user delegates her trust explicitly to another party (which can be a server or another user), which is identified via a DN. The trusted party can then work on the user's behalf, even delegate trust again, forming a trust delegation chain. For more details we refer the reader to the more extensive description in [12].

When the user does not have a X.509 private key, she cannot sign any assertions. Thus, in UNICORE 7 the Unity server can be used as the source of the “bootstrap” trust delegation that asserts that the user trusts the first server. As the trust delegation mechanism is based on SAML and requires heavy weight XML processing, it does not readily lend itself to a RESTful architecture, especially it has to be avoided on the client side.

In delegation, two use cases need to be considered:

1) REST-to-REST : the user invokes a RESTful service, which needs a delegated call to another RESTful service
2) REST-to-SOAP : user invokes a RESTful service, which needs a delegated call to a SOAP service

IV. First results

We have implemented a number of extensions to link the JAX-RS implementation provided by Apache CXF to the UNICORE resources framework provided by USE. These include

- an authentication handler uses the HTTP basic authentication header (i. e., username and password) and maps them to a X.500 DN using a configurable chain of authentication components. Available authentication options are a local username/password file or the admin can delegate the authentication to Unity;
- mechanisms are used to inject the requested resources and other required information into the JAX-RS service class;
- access control checks using the XACML policy decision point.

As a baseline, we have started the implementation of RESTful services for the fundamental UNICORE entities listed in Section II.

Both JSON and HTML representations of individual resources can be served, as well as lists of all the resources available to the current user. Table I shows the current state of the API.

TABLE I. Initial REST API for UNICORE services.

HTTP method on resource	*Description*	*Media type*
GET /jobs	Lists user’s jobs	JSON, HTML
POST /jobs	Submits a new job	JSON
GET /jobs/{id}	Get job properties	JSON, HTML
DELETE /jobs/{id}	Remove a job	
GET /storages	Lists user’s storages	JSON, HTML
GET /storages/{id}	Get storage properties	JSON, HTML
DELETE /storages/{id}	Remove a storage	
GET /storages/{id}/files/{path}	Get file properties	JSON, HTML
GET /storages/{id}/files/{path}	Download a file	Binary
PUT /storages/{id}/files/{path}	Upload a file	Binary
POST /storages/{id}/imports	Create a new file import	JSON
POST /storages/{id}/exports	Create a new file export	JSON
GET /sites	Lists user’s sites	JSON, HTML
GET /sites/{id}	Get site properties	JSON, HTML
DELETE /sites/{id}	Remove a site	

It is possible to consistently access these resources through both the WSRF layer (using standard UNICORE clients) and through the REST layer (using HTTP clients, such as *curl*).

Data can be downloaded and uploaded through the web server using the HTTP protocol using GET and PUT requests. In addition, other file transfer protocols (e.g., UFTP) are supported as well by explicitely creating new file import/export resources.

The services API is currently under discussion and further development. The UNICORE resources form a tree with many interconnections. For example, a job has a working directory, which is a storage resource. Thus, the working directory resource should be accessible via both `/jobs/{j_id}/wd` and `storages/{s_id}`. In the WSRF API, these links between resources are discovered by the client, and in RESTful designs this dynamic discovery of resource links is considered the most elegant (according to the “HATEOAS” principle in [7]). On the other hand, having to dynamically discover everything can lead to increased network traffic and latencies, and clients may want to leverage certain knowledge of the REST API.

The delegation issue is solved partly: when authenticating a user using Unity, the REST authentication handler also receives a SAML assertion, which can be used later to make invoke services on behalf of the user. However, this only works when invoking SOAP/WSRF services. A solution for delegated access to resources through the REST API still needs to be agreed upon. Several possible solutions are conceivable. For example, a JSON rendering of the SAML assertions used by UNICORE is possible. Since these would be handled entirely on the server, the clients would not be made more complex.

A. Job submission example

For job submission, a simple JSON job description is used, which consists of the executable, arguments, environment settings as well as data stage-in and stage-out and required consumable resources such as wall time. This is currently translated into UNICORE’s internal XML format before being submitted to the internal execution engine. As a trivial example,

```
{
  Executable: "/bin/echo",
  Arguments: ["Hello World"],
}
```

would be a valid job. This JSON job description syntax is already in use in the UNICORE commandline client [13], and thus well known to UNICORE users. Using curl as a simple HTTP client, the submission of a job in file “job.u” can be done by (ignoring security for the moment):

```
  curl -X POST <base_url>/jobs
    -H "Content-type: application/json"
   --data-binary @job.u -i
```

The server will reply with a “201 Created” status and the location of the new job:

```
HTTP/1.1 201 Created
Location:  <base_url>/jobs/<id>
```

B. Initial performance tests

Since the new REST interface shares the back-end and business logic with the WSRF interface, any performance improvements are due to the smaller overhead of the REST interface. To quantify these improvements, we have run a number of simple performance tests, comparing a “GET” operation on a resource via both the WSRF and the REST interfaces. We have used a production-like setup, where the REST service is accessed via SSL and via a UNICORE Gateway. All servers and the client code was run on “localhost”. The test machine was a quad-core Intel i7 at 2.8GHz, with 8 GBs of RAM running Java 7 (OpenJDK 1.7.0_65).

TABLE II. THROUGHPUT FOR GET REQUESTS VIA WSRF AND REST.

Client threads	*Interface*	*Requests/sec*
1	WSRF	27
	REST	79
2	WSRF	57
	REST	193
4	WSRF	80
	REST	286
8	WSRF	76
	REST	332

We sent 1000 requests each using 1,2,4 or 8 client threads. As table II shows, using the REST interface has much higher throughput, and scales better to higher numbers of concurrent client threads.

As a second example, we have evaluated job submission, using simple 'hello world' jobs as shown above. Here we submitted 400 jobs. Table III shows the results. Again the REST interface is onsistently better in terms of throughput and scalability.

TABLE III. THROUGHPUT FOR JOB SUBMISSION VIA WSRF AND REST.

Client threads	*Interface*	*Jobs/sec*
1	WSRF	5
	REST	34
2	WSRF	11
	REST	54
4	WSRF	12
	REST	75

These initial tests already show that significant performance improvements can be expected from the REST interface.

V. SUMMARY AND OUTLOOK

We have extended UNICORE to allow building RESTful services that are fully consistent with the existing SOAP/WSRF based services. This includes the security stack used for RESTful services, which is fully compatible and consistent with the rest of the UNICORE world.

One fundamental issue remains to be fully solved: delegation to allow a server to make delegated calls to other RESTful services. One option is to use Unity to provide SAML assertions once the user has authenticated, and translate the SAML delegation assertions to a JSON rendering.

The new RESTful APIs will open up the world of HPC and access to large-scale scientific data to a much wider audience, by allowing applications to use simple authentication mechanisms, submit compute tasks to HPC machines, manage results, move data and much more.

The basic REST support and initial service implementations will be released with UNICORE 7.1, and will be evolved further towards a major release, UNICORE 8.

Next steps will focus on finalizing the security architecture and implementing OpenID-Connect support, i. e. validating OIDC tokens and if required creating SAML trust delegation assertions from them using Unity. Furthermore, the service APIs will be developed further, aiming at basic job submission and management for the first release, and adding full capabilities in the UNICORE 8 release.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 604102 (Human Brain Project)

REFERENCES

[1] “UNICORE Website,” http://www.unicore.eu/, [accessed: 2014-07-10].

[2] “UNICORE Open Source project page,” http://sourceforge.net/projects/unicore/, [accessed: 2014-07-10].

[3] “Web Services Resource Framework,” http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf, [accessed: 2014-07-10].

[4] “Human Brain Project,” http://www.humanbrainproject.eu/, [accessed: 2014-07-10].

[5] “OpenID Connect,” http://openid.net/connect, [accessed: 2014-07-10].

[6] B. Schuller and T. Pohlmann, “UFTP: High-Performance Data Transfer for UNICORE,” in Proceedings of 7th UNICORE Summit 2011, ser. IAS Series, no. 9. Forschungszentrum Jülich GmbH, 2011, pp. 135–142.

[7] R. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000.

[8] D. Crockford, “The application/json media type for javascript object notation (JSON),” RFC 4627, Jul. 2006.

[9] B. Schuller, R. Menday, and A. Streit, “A Versatile Execution Management System for Next-Generation UNICORE Grids,” in Proceedings of 2nd UNICORE Summit 2006 in conjunction with EuroPar 2006, ser. LNCS, no. 4375. Springer, 2006, pp. 195–204.

[10] “Java API for RESTful Services (JAX-RS),” https://jax-rs-spec.java.net/, [accessed: 2014-07-10].

[11] “Unity Identity Management Solution,” http://www.unity-idm.eu/, [accessed: 2014-07-10].

[12] K. Benedyczak, P. Bała, S. van den Berghe, R. Menday, and B. Schuller, “Key aspects of the UNICORE 6 security model,” Future Generation Computer Systems, vol. 27, 2011, pp. 195–201.

[13] “UNICORE commandline client job description format,” http://unicore.eu/documentation/manuals/unicore6/files/ucc/ucc-manual.html#ucc_jobdescription, [accessed: 2014-07-10].