NORDUGRID-TECH-19
22/10/2010

The Hosting Environment of the Advanced Resource Connector middleware

D. Cameron, M. Ellert, J. Jönemo, A. Konstantinov*, I. Marton, B. Mohn, J. K. Nilsen, M. Nórden, W. Qiang, G. Roczei, F. Szalai, A. Wäänänen

*aleksandr.konstantinov@fys.uio.no

Contents

Chapter 1

Introduction

The Hosting Environment Daemon (HED) is the container of all the functional components of the next generation of the Advanced Resource Connector (ARC) middleware on the server side. It is the central part in a new very lightweight incarnation of ARC that is aimed at - but not limited to - providing Web Service.

The whole design of the HED is built around the idea of flexibility and modularity. Inside HED the developer or deployer is supposed to use the minimum amount of components and external dependencies only. This is why the HED mostly consists of pluggable modules with some glue among them.

Because in its current state it mostly provides modules for building SOAP based Web Services, it is easy to think that HED is just another Web Services development framework like Axis, gSOAP, XFire or any other out of the numerous implementations. Instead, the idea of HED is to provide framework for gluing functionalities and not a re-implementation of various standards. Effectively that means if Apache 2 web server is considered by developers as necessary for serving as frontend to services there could be plugin written which places Apache 2 into a chain of other plugins of the HED.

In the current implementation there are no Apache or Axis plugins. That is because the developers of HED were very much concerned about making the solution lightweight and needed an implementation of those supported protocols that was both simple and lightweight. As a result essentials like SOAP and HTTP are implemented inside HED, while external software is used whenever was appropriate - as in the case of TLS, (Grid)FTP, LDAP and some other cases. That does not exclude possibility to have plugins using entirely external solutions either developed or accepted from third parties.

The HED is a relatively young framework and there are quite a few rough edges and non-flexible solutions. The situation will hopefully improve with time. We would be grateful for any constructive comments and suggestions how to improve the architecture and the code of the HED.

Chapter 2

How to read

This document does not (yet?) include in depth description of C++ classes which constitute the HED. Instead most sections contain notes entitled "Relevant classes". Technical description of those classes can be found in automatically generated "Hosting Environment (Daemon) Reference Manual" document [**?**].

For examples please see the source code in the repository [**?**]. Many components and libraries are accompanied with test and example applications.

Chapter 3

Architecture

3.1 Requirements

In the design of the HED, several goals and requirements were considered. These were weighed against each other and the factual context.

The implementation language had to be object oriented, efficient and capable of providing easy access to system functionality. This eventually lead to the adoption of C++. But languages such as Java and Python were also considered in the early stage. Our investigation showed that there are Open Source tools which allow C++ modules to be relatively easy interfaced from modules written in other considered programming languages. C++ modules also work in opposite way being capable to use modules written in various other languages.

External dependencies needed to be kept to a minimum while also taking into consideration their ubiquity or relative rarity as well as license related concerns. Software of this level of complexity must depend on many external libraries and components but each such dependency has been introduced only after thorough consideration.

Using resources in efficient way was an important goal. The present design enables many services sharing both the same process and the same network port(s) while exhibiting a remarkably low memory footprint.

3.2 Technical design

In the technical design it turned out that providing dynamic loading, portability and a well tested high level memory management could all be greatly assisted by introducing glibmm [**?**] - the C++ interface made for the GNOME [**?**] projects library for memory management and related functionality. This enables the developers to write code easily portable across various operating systems and architectures.

The term HED means three things:

1. the daemon (called `arched`) which hooks up the system and initialize components the way as it is described in the configuration files. This configuration describes the components and their relations to each other. In optimal cases these single services run on any node where ARC1 is deployed and started. Without loadable components the daemon itself does nothing usefull.

2. sometimes using the HED terms to refer to collection of libraries which is used by service or other component developers. These libraries define interfaces and implement some common classes which may simplify the life of service and component developers however only few of these classes are mandatory to use to make the components and services loadable and hookable by the daemon.

3. the collection of components implementing minimal set of protocols needed for implementing so called Web Services.

   Unless otherwise stated the term HED will be used through this document to refer to the second option - framework of C++ classes.

3.2.1 MCC

`Relevant classes: Arc::MCC, Arc::Loader`

In the HED data channels to the outside world may be set up by chains of small processing units called Message Chain Components (MCCs). The chain is an ordered list of MCCs and their interconnection can be described in the configuration file. The MCCs work on units called Messages which represent data going in to or out of the HED. The Message consists of the so called Payload which is its main content structured in a way relevant to the protocol of the corresponding MCC, and auxiliary structures such as general Attributes and Security Attributes where information relevant to each protocol is accumulated as the Message progresses. Each MCC typically implements one level in the Internet Protocol suite by transforming the Message to an input suitable to propagate to the next component and then performs the corresponding transformation of the response on the way back. The components are all dynamically loaded to provide maximum flexibility and extensibility. Each instance of these MCC's can be individually configured.

Each MCC has an entry method `process()` which is called with Message being processed. It then processes Message by modifying it or creating new Message. Then MCC calls entry method of the next MCC in the chain. For information how Messages are handled and about memory management policies please see API description of MCC class in [**?**].

The developer who writes an MCC is free to choose any 3rd party library and component to implement the functionality of the MCC but at least currently the MCCs should be written in the same language as the HED was written (C++) and should use the `MCC` interface class and `Message` class provided by `arcloader` and `arcmessage` libraries of ARC1.

The MCC may implement some routing algorithm which means one MCC may have connections to multiple other MCCs. Typical scenario is that the HTTP MCC at the server side routes the HTTP Messages with POST HTTP operation to a SOAP MCC but the Messages with GET operation to for example a simple HTTP service component. For that purpose ¡next¿ elements in MCC configuration may have optional `id` attribute which allows to assign labels to all chain links to the next MCCs in the chain. Supported labels are MCC dependent. By default simple MCCs support only one unnamed link. The MCCs with routing capabilities must have all supported labels documented.

As the data is passed through the individual MCCs, they each populate structures with both general attributes and special security attributes that are available at that particular protocol level.

In general every MCC has optimal and natural places in certain chains and this place cannot always be modified. For example on the server side the TCP MCC must be the first MCC in any chain where it is used and the TLS MCC should be right after the TCP MCC.

Server and Client Side MCCs

Most of the MCCs has a client and a server version because the behavior of an MCC should be different depending on whether it is sitting on the server or client side. The typical scenario here is illustrated by the TCP MCC which should listen and wait for incoming messages on a socket on the server side but call `connect()` on the client side.

Server and Client side MCCs are separate elements although definitely sharing some code and normally provided inside the same plugin module.

Payload

`Relevant classes: Arc::Payload, Arc::PayloadRawInterface, Arc::PayloadRaw, Arc::PayloadStreamInterface, Arc::PayloadStream, Arc::PayloadSOAP`

Main content of the information is transferred using the Payload part of the Message. There are no limitations on functionality of Payload object except that it must be inherited from MessagePayload class. Despite being flexible such approach would be useless. This is why HED defines three Payload interfaces and their simplest implementations. All MCCs which are distributed with the HED use, implement and extend those interfaces. Those include:

1. PayloadRawInterface and its implementation PayloadRaw. This interface represents set of catenated in-memory chunks. It's meant to be used for information available as whole. And also for prepending and appending information without actually moving and copying data chunks in memory.

2. PayloadStreamInterface and its implementation PayloadStream. It covers case of sequentially accessible information. The main purpose of that Payload is to serve protocols which define continuous data stream like TCP.

3. PayloadSOAP represents parsed SOAP message. It's introduced to cover need for writing SOAP based Web Services in unified way.

Each MCC developed inside and outside the HED must be accompanied with description of the Payloads it supports on input and those generated on output. Those types should be taken into account while creating chains of the MCCs. There are no Payload type checks done during the chain configuration phase. Hence Payload incompatibility problems will be detected only during runtime.

Attributes

`Relevant classes: Arc::MessageAttributes, Arc::Message`

The Message object may contain general purpose key and value pairs called Attributes. Keys and values are simple strings. Each key may have multiple corresponding values. All pairs are handled by MessageAttributes class. Codewise there are no limitations posed on content and purpose of Attributes.

By convention keys are composed of two parts: name of MCC/Protocol at which Attribute was generated or must be consumed and name of Attribute itself separated by column. For example the Attribute with key HTTP:METHOD holds HTTP protocol method like GET, PUT, HEAD, etc. It is either generated by MCC implementing HTTP protocol or is filled by other code and is used by HTTP MCC to generate proper HTTP header.

Each MCC developed inside and outside the HED must have generated and consumed Attributes described in accompanying documentation.

Security Attributes

`Relevant classes: Arc::SecAttr, Arc::MessageAuth, Arc::MessageAuthContext, Arc::Message`

Here only basic information about security related objects is presented. For more detailed information please see "Security Framework of ARC1" [**?**].

Security Attributes are storing various aspects of Message useful for authorization decisions to be made. Those normally include the operation being requested, target of operation and identity of subject making request. They can also contain authorization policies. Actually nothing stops from storing an other type of information but there is no convention developed for that.

The Security Attributes are stored as key and value pairs. Each key may have only single value attached. Keys are simple strings. By convention each MCC or Security Handler (see below) produce Security Attribute with name corresponding to protocol name. For example Security Attribute stored under name TLS holds information collected at Transport Level Security layer.

The value of Security Attribute is an object of class inherited from SecAttr. The HED implements class SecAttr which serves as definition of interface for all Security Attributes. It defines the way how collected information may be turned into format suitable for making authorization decisions. For that each Security Attribute value implements method Export for converting internal information into one of the supported formats. Currently only implemented is ARC Authorization Request/Policy (see below). Please see API description of SecAttr and MessageAuth classes in [**?**] for more information.

Each MCC developed inside and outside the HED must come with explanation of generated and consumed Security Attributes. Dedicated components for dealing with Security Attribute - Security Handlers are described below.

Security Handlers

`Relevant classes: ArcSec::SecHander, Arc::MCC`

Each MCC can also be configured to have loadable modules called Security Handlers attached to it in order to enforce security policies such as authentication and authorization or to assist such activities by gathering specialized security related information into Security Attributes. There is no strict distinction of capabilities between Security Handlers and MCCs. Both can and do populate Security Attributes. The distinction is more of logical nature. It also makes possible to have Security Handlers dealing with similar kind of information and capable of acting on different protocol levels.

The Security Handlers are arranged into named queues. Elements in every queue are executed sequentially with the Message as the only argument. Different queues are executed at different times. Which queue is executed in which case it depends on the MCC. Most MCCs implement two queues named "incoming" and "outgoing". The queue "incoming" is executed for Messages moving through the chain towards hosted application and passes the Message as argument with Payload of type corresponding to MCC type i.e. HTTP MCC passes Payload with parsed HTTP message. The "outgoing" queue is executed for Messages travelling to outside the HED.

3.2.2 Services and Clients

The services are dynamically loaded on start up just like the MCCs. They are almost identical to MCCs with an exception that they constitute the last link in the Message Chain. They are attached to the Chain in the same manner as other MCCs. But differently from MCC the `process()` method of the Service does not pass Message to other components of the Chain. Instead they have to process incoming Messages and produce outgoing ones.

The clients are not represented by any specific component. The client code sees the Chain as single object with named entry points. Those entries are used by clients to insert request Message and get result Message on output. For simplifying the task of writing clients there is a library `arcclient` provided which wraps the task of creating Chains and Messages for the widely used SOAP, HTTP, TLS, TCP communications.

3.2.3 Plexer

`Relevant classes: Arc::Plexer`

In general case multiple Services are living in the HED so the incoming Message should be routed to the proper Service. The Plexer MCC does this job. It takes the ENDPOINT attribute of the Message collected by other MCCs compares this attribute to regular expression defined in the configuration file and forwards the Message to the matching service. It acts as a dispatcher. The Plexer is also special in a sense that it is not a plugin but part of the `arcloader` library.

The Plexer provides only basic functionality and is capable of doing only simple routing. But because each MCC has multiple routing capabilities it is possible to provide pluggable MCC implementing more sophisticated and/or more specific routing algorithms.

3.2.4 Error handling

`Relevant classes: Arc::MCC_Status`

For reporting errors each `process()` method returns instance of MCC_Status class. That object carries predefined set of common error codes.

This way for error reporting is mostly meant to be used for reporting problems related to code execution. For errors caused by processing the corresponding protocol MCC should generate proper response Message which carries error description. Only if protocol does not provide error handling MCC_Status should be used. It is also advised to convert MCC_Status error obtained from the next MCC in the Chain into protocol specific error Message if possible.

3.2.5 Instantiation of the Chain

`Relevant classes: Arc::Loader, Arc::Config, Arc::LoaderFactory, Arc::MCCFactory, Arc::ServiceFactory, Arc::DMCFactory, Arc::SecHandlerFactory, Arc::PDPFactory, Arc::ACCFactory`

Chain instantiation is handled by Loader component. It takes XML configuration on input and then handles tasks of loading plugin libraries, identifying plugins in them, creating and linking objects of corresponding classes.

Each Loader object may create multiple non-intersecting chains and there may be multiple Loader objects in same executable. For each component mentioned in the configuration the Loader creates single object of the corresponding class. On Loader destruction all handled components are destroyed too.

The configuration of Loader is made up of the following elements (see also configuration schema at http://svn.nordugrid.org/trac/nordugrid/export/10146/arc1/trunk/src/hed/libs/loader/mcc.xsd):

- `Chain` is base element. It has no special meaning for configuration of chain and is only used to group other elements in a logical way. Chains may be nested.
- `Component` represents the MCC. Its attribute `name` defines the name of the MCC which has to be instantiated. This MCC is being looked up among those contained in modules loaded by `ModuleManager` element. If not found then the Loader tries to find a module with name corresponding to name of the MCC. Each MCC also must have unique identifier specified by `id` attribute. Subelement `next` specifies link to the next MCC in the Chain. It uses unique identifier specified by `id`. The `next` element may have an optional text which is used to distinguish different Message propagation path in MCC specific way. The `SecurityHandler` subelement defines the Security Handler and the queue to which it has to be attached. Also attribute `entry` may be used to create an entry point to the Chain.
- `Plexer` element instantiates Plexer component. Its configuration is similar to one of the `Component` elements.
- `Service` element defines plugin implementing final Component in the Chain - service. From Loader point of view its configuration is identical to that of the `Component`.
- `ModuleManager` defines parameters needed to find loadable modules containing plugins. Currently only subelement `path` is supported which defines path on file system where loadable modules may be found.
- `Plugins` specifies name of the loadable module.

3.2.6 Sessions and Contexts

`Relevant classes: Arc::MessageContext, Arc::MessageContextElement, Arc::MessageAuthContext, Arc::ChainContext`

The HED defines three lifetimes for operations happening inside the Chain and components which can be associated with them:

1. The Message lifetime - lasts as long as incoming and outgoing Messages are passing through the Chain forth and back. The Message itself in this case is used as container for associated components.
2. The Session lifetime - is defined by existence of some logical connection between Messages being processed. Corresponding container MessageContext is normally created by the first MCC in a chain and attached to the Message. Actual lifetime of that container is MCC specific. For TCP MCC it corresponds to TCP connection. The MessageContext holds objects inherited from the MessageContextElement class. The Security Attributes can also be stored in dedicated container - MessageAuthContext - with Session lifetime. At the end of the Session all associated objects are destroyed.
3. The Chain lifetime - is the time period while component that made the Chain exists. This lifetime is represented by ChainContext class. Differently from other context objects this one does not allow free manipulation of contained objects. Instead, it provides an interface to some internal structures of the Loader object. Those include factories and lists of objects of all types created by particular instance of the Loader.

3.2.7 DMC

`Relevant classes: Arc::DataPoint, Arc::DataMover, Arc::DataBuffer and related classes.`

The HED defines an interface for pluggable components implementing higher-level information transfer and query. Those are Data Management Components (DMC). Each DMC is inherited from Datapoint class and provides subset of methods for performing the following operations on data endpoint:

1. Read data from specified endpoint into DataBuffer class object
2. Write data into specified endpoint from DataBuffer class object
3. List subcontent of endpoint (i.e. list files in directory)
4. Register and unregister presence of data - for indexing endpoints
5. Resolve final or lower level location of data from stored metadata - for indexing endpoints

The DMC may be implemented using third party software like it is currently done for (Grid)FTP DMC. But implementation may use Message Chains too like in case of HTTP MCC.

Along with ordinary endpoints defining the location of data directly - like HTTP, FTP, LDAP - the DMC can be used with indirect/indexing endpoints. Those are endpoints which define only the location of meta-data associated with actual data or an interface/service providing functionality of managing/requesting data. For more information about indexing endpoints see description of supported URLs in [**?**].

More in depth technical information about the DMC can be found in [**?**].

3.2.8 Generic purpose components

The HED also includes vast amount of common purpose components:

- XMLNode is class for managing parsed XML structures. It provides minimal functionality for operating on XML elements, attributes and namespaces.
- Thread management functions provide few mostly used functions for thread creation.
- URL class gives access to various parts of URL. It has support for many specific URLs built-in.
- String utilities for easy conversion between various types to strings and back.
- Run class for starting, communicating and monitoring external processes.
- Logger class provides multilevel controllable logging functionality.
- Time and Period classes allows parsing and generating textual description of times, dates and time periods in various formats.
- RegularExpression class providing C++ wrapper for regexp related functions.
- Config class makes it possible to manipulate configuration file. And related classes give an access to specific parts of configuration.
- Counter and IntraProcessCounter provide a way to count abstract resources.

3.2.9 Web Service related components

There is also a set of components implementing various Web Service related functionality. Those include:

- WSAEnpointReference and WSAHeader manipulate WS-Addressing [**?**] information in XML element and SOAP header correspondingly.

- SAMLToken, UsernameToken and X509Token are for consuming and generating token of same name according to various profiles of WS Security specifications [**?**].
- WSRF and related and inherited classes offer a way to generate and analyze various Web Services Resource specification [**?**] related elements.
- DelegationConsumerSOAP, DelegationProviderSOAP and DelegationContainerSOAP implement web service interface which enables client to delegate X509 credentials to service.
- InformationInterface and related classes may be used to implement information interface service and client part in a way common for services built on top of HED. Also InfoCacheInterface extends it with caching functionality and InfoRegister provides service registration ability.

3.2.10 Daemon

All componets of the Hosting Environment are compiled into loadable libraries and may be used in various executables. But there is also a dedicated executable provided - `arched`. It accepts configuration file containing XML document and passes it to the Loader component. Then the Loader component takes care of loading all modules and instantiating all Message Chains. The `arched` also initializes the Logger component, configures it and directs its output to where it is specified or to the standard output. The `arched` normally runs as background process but can be run in foreground as well. For more information about `arched` capabilities please read its manual page.

3.2.11 Alternative implementation languages

In order to facilitate the development of services, API bindings for languages other than C++ are provided and some service development has already been done in Python language. Currently the only available language bindings are Python and Java. Currently it is possible to write only SOAP Service modules in Python and Java due to multiple inheritance limitation.

Chapter 4

Implemented elements

This chapter describes components which are implemented alongside with the HED infrastructure. Although strictly not belonging to the infrastructure this minimal set of components is necessary to make the infrastructure usable.

4.1 Implemented MCCs

4.1.1 TCP MCC

`Plugin names: tcp.service, tcp.client`

`Library name: {lib}mcctcp`

`Security handler queues: incoming, outgoing`

`Message attributes: TCP:HOST, TCP:PORT, TCP:REMOTEHOST, TCP:REMOTEPORT, TCP:ENDPOINT, ENDPOINT`

The server side TCP MCC in the HED is special in that it produces Messages by listening on a network socket rather than processing Messages from other MCCs. As such it spawns new thread for every new connection to handle Messages and their responses throughout the Message Chain. One could envision other MCCs having these properties but producing Messages from other sources such as e.g. UNIX sockets.

This MCC can be configured with one or more `<tcp:Listen>` elements which in turn contain the elements `<tcp:Port>`, `<tcp:Interface>` and `<tcp:Version>`. The `<tcp:Port>` element is mandatory and should contain an integer corresponding to the TCP port to listen to. The `<tcp:Interface>` element is optional and is meant to identify the network interface to bind to. It is currently not used. The `<tcp:Version>` element is used to specify IP version. It is optional and should if present contain the single digit 4 or 6.

The server side TCP MCC generates `PayloadStreamInterface` payload in the Message passed to the next MCC which can be used for communicating through open TCP channel. Currently it ignores any payload attached to the returned Message. That MCC also fills following Message Attributes in the incoming Message while passing to the next MCC:

1. TCP:HOST - IP address of local interface which was used to establish TCP connection
2. TCP:PORT - local TCP port which was used for connection
3. TCP:REMOTEHOST - IP address of contacting client
4. TCP:REMOTEPORT - TCP port of contacting client
5. TCP:ENDPOINT - URL-like combination of ://(TCP:HOST):(TCP:PORT)
6. ENDPOINT - same as TCP:ENDPOINT

The client side TCP MCC performs TCP connection to host and port specified in ¡Host¿ and ¡Port¿ elements inside ¡Connect¿ element of the MCC configuration. Then all incoming Messages of `process()` method are transferred over TCP connection. Accepted Payload type of incoming Message is `PayloadRawInterface`. Returned Payload is of `PayloadStreamInterface` type. It represents established TCP connection and `may` be used by previous MCCs in chain for direct communication. It is still preferred to call `process()` method instead.

Configuration schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.nordugrid.org/schemas/ArcMCCTCP/2007"
  xmlns:arc="http://www.nordugrid.org/schemas/ArcConfig/2007"
  targetNamespace="http://www.nordugrid.org/schemas/ArcMCCTCP/2007"
  elementFormDefault="qualified">

    <xsd:simpleType name="Version_Type">
        <!-- This element defines TCP/IP protocol version.  -->
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="4"/>
            <xsd:enumeration value="6"/>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:element name="Version" type="Version_Type"/>

    <xsd:complexType name="Listen_Type">
        <!--
            This element defines listening TCP socket. If interface is missing socket
            is bound to all local interfaces (not supported). There may be multiple Listen elements.
        -->
        <xsd:sequence>
            <xsd:element name="Interface" type="xsd:string" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="Port" type="xsd:int" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="Version" type="Version_Type" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="Listen" type="Listen_Type"/>
    <xsd:complexType name="Connect_Type">
        <!--
            This element defines TCP connection to be established to specified Host at specified Port.
            If LocalPort is defined TCP socket will be bound to this port number (not supported).
        -->
        <xsd:sequence>
            <xsd:element name="Host" type="xsd:string" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="Port" type="xsd:int" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="LocalPort" type="xsd:int" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="Connect" type="Connect_Type"/>
</xsd:schema>
```

4.1.2 TLS MCC

`Plugin names:` `tls.service, tls.client`

`Library name:` `{lib}mcctls`

`Security handler queues: incoming, outgoing`

`Message attributes: TLS:PEERDN, TLS:IDENTITYDN`

The server and client TLS MCCs provide transport level security (TLS) over any stream channel. Currently they interoperate well with TCP MCCs.

The server side MCC accepts payload of type PayloadStreamInterface. It then creates own instance of object inherited from PayloadStreamInterface bound to initial payload and passes it to the next MCC. This object is maintained inside Message Context under name `tls.service` and is destroyed when Context becomes inactive. Currently this MCC does not expect any payload to be returned from the rest of the chain and passes no payload to previous MCC.

The server MCC fills Message Attributes TLS:PEERDN and TLS:IDENTITYDN representing subjects of last certificate in client's certificate chain used for establishing secure connection and the subject of last certificate which is not a proxy certificate correspondingly. Use of those attributes is deprecated. It is advised to use Security Attributes instead.

The client side MCC behaves in similar way. It also establishes PayloadStreamInterface type object linked to same type of payload of the next MCC. To obtain that last payload it makes a first call to the next MCC with payload of type PayloadRawInterface and then uses the returned payload - which is expected to be of PayloadStreamInterface type - to create its own payload object with streaming capabilities and returns it to previous MCC for further usage.

Both client and server side MCCs are implemented using OpenSSL toolkit [**?**] and use X.509 infrastructure [**?**] for establishing secure connection and may be configured to get private key, certificate or proxy credentials from files residing at local file systems. It is also possible to specify the location of Certification Authority certificate or to use all certificates located in specified directory. for more information see configuration schema with comments below.

Configuration schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.nordugrid.org/schemas/ArcMCCTLS/2007"
  xmlns:arc="http://www.nordugrid.org/schemas/ArcConfig/2007"
  targetNamespace="http://www.nordugrid.org/schemas/ArcMCCTLS/2007"
  elementFormDefault="qualified">
    <xsd:complexType name="CACertificatesDir_Type">
      <xsd:simpleContent>
        <xsd:extension base="xsd:string">
          <xsd:attribute name="PolicyGlobus" type="xsd:boolean" use="optional" default="false"/>
        </xsd:extension>
      </xsd:simpleContent>
    </xsd:complexType>
    <!-- Location of private key.
      Default is /etc/grid-security/hostkey.pem for service
      and none for client. -->
    <xsd:element name="KeyPath" type="xsd:string"/>
    <!-- Content of private key - not supported -->
    <xsd:element name="Key" type="xsd:string"/>
    <!-- Location of public certificate.
      Default is /etc/grid-security/hostcert.pem for service
      and none for client. -->
    <xsd:element name="CertificatePath" type="xsd:string"/>
    <!-- Content of public certificate - not supported -->
    <xsd:element name="Certificate" type="xsd:string"/>
    <!-- Location of proxy credentials - includes certificates, key and
      chain of involved certificates. Overwrites elements Key, KeyPath,
      Certificate and CertificatePath.
```

```
      Default is none for client and none for service. -->
    <xsd:element name="ProxyPath" type="xsd:string"/>
    <!-- Content of proxy credentials - not supported -->
    <xsd:element name="Proxy" type="xsd:string"/>
    <!-- Location of certificate of CA. Default is none. -->
    <xsd:element name="CACertificatePath" type="xsd:string"/>
    <!-- Content of certificate of CA - not supported -->
    <xsd:element name="CACertificate" type="xsd:string"/>
    <!-- Directory containing certificates of accepted CAs.
      Default is /etc/grid-security/ . -->
    <xsd:element name="CACertificatesDir" type="xsd:string"/>
</xsd:schema>
```

4.1.3 HTTP MCC

`Plugin names: http.service, http.client`

`Library name: {lib}mcchttp`

`Security handler queues: incoming, outgoing`

`Message attributes: HTTP:METHOD, HTTP:CODE, HTTP:REASON, HTTP:RANGESTART, HTTP:RANGEEND, HTTP:ENDPOINT, HTTP:*, ENDPOINT`

The server side HTTP MCC accepts Messages with `PayloadStreamInterface` payload and parses HTTP related information from it. Information from the HTTP header is added to the Message Attributes. The body of HTTP message is passed to the next MCC as `PayloadRawInterface` payload. In response this MCC expects also the Message with `PayloadRawInterface`. It is then prepended with HTTP response header and pushes it into initially provided stream channel. As an output it returns empty `PayloadRawInterface` payload.

That MCC routes results to multiple next MCCs in the chain. For that it accepts only labeled ¡next¿ elements in the configuration. Label names are those of HTTP methods (uppercase). HTTP Messages will be routed to their destinations according HTTP method requested.

That MCC also fills following Message Attributes in the incoming Message while passing to the next MCC:

- HTTP:METHOD - HTTP method as defined in HTTP request header.
- HTTP:RANGESTART - Range request start offset.
- HTTP:RANGEEND - Range request end offset.
- HTTP:ENDPOINT - URL or path as specified in HTTP request header.
- HTTP:* - Here * stands for any name. All HTTP options from HTTP request header are converted into Message Attributes named HTTP:{option name}.
- ENDPOINT - same as HTTP:ENDPOINT.

For outgoing Message server MCC converts all HTTP:* Message Attributes into corresponding HTTP response header attribute.

The client side HTTP MCC will accept `PayloadRawInterface` payload as HTTP body and after prepending it with HTTP information passes to the next MCC also as `PayloadRawInterface`. It accepts `PayloadStreamInterface` in response and after processing passes `PayloadRawInterface` back through the chain.

The client MCC also converts all HTTP:* Message Attributes into corresponding HTTP header options while creating HTTP request. If defined it uses HTTP:METHOD and HTTP:ENDPOINT as method and URL/path of HTTP request. If not specified parameters defined in the configuration are used. For HTTP response it sets HTTP:CODE and HTTP:REASON to response code and reason correspondingly. It also performs conversion from header options into HTTP:* Message Attributes.

Configuration schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.nordugrid.org/schemas/ArcMCCHTTP/2007"
  xmlns:arc="http://www.nordugrid.org/schemas/ArcMCCHTTP/2007"
  targetNamespace="http://www.nordugrid.org/schemas/ArcMCCHTTP/2007"
  elementFormDefault="qualified">

    <!--
        These elements define endpoint and HTTP method for client HTTP MCC.
    -->
    <xsd:element name="Endpoint" type="xsd:string"/>
    <xsd:element name="Method" type="xsd:string"/>
</xsd:schema>
```

4.1.4 SOAP MCC

`Plugin names: soap.service, soap.client`

`Library name: {lib}mccsoap`

`Security handler queues: incoming, outgoing`

`Message attributes: SOAP:ENDPOINT, ENDPOINT`

These MCCs convert payloads between data chunks presented by PayloadRawInterface type object into dedicated PayloadSOAP payloads and vice versa. Currently it has no specific configuration parameters.

The server side MCC sets Message Attributes SOAP:ENDPOINT and ENDPOINT to URL present in `To` element of Web Service Addressing information stored in SOAP header. For proper SOAP over HTTP binding it also sets `HTTP:Content-Type` Message Attribute to value `application/soap+xml` or `text/xml` for SOAP version 1.2 and 1.1 respectively for response Message. Also in case of SOAP fault response `HTTP:CODE` is set to `500` and `HTTP:REASON` to `SOAP FAULT`.

The client side MCC for outgoing Message accepts `SOAP:ACTION` Message Attribute and uses it to fill `HTTP:Content-Type` or `HTTP:SOAPAction` depending on SOAP version.

4.2 Implemented Security Handlers

For more information about security related capabilities of the HED please see dedicated document at [**?**].

4.3 Implemented DMCs

4.3.1 File DMC

`Protocol name: file`

This DMC implements access to local or remote mounted file systems. It allows reading and writing the content of file, listing the content of directories and checking the presence of the object.

Name of plugin is `file` and it is located in {`lib`}`dmcfile` loadable module.

4.3.2 GridFTP/FTP DMC

`Protocol name: ftp, gsiftp`

This DMC implements access to a data server using FTP or GridFTP protocol. It allows reading and writing the content of stored file, listing the content of directories and checking the presence of the object.

Name of plugin is `gridftp` and it is located in {`lib`}`dmcgridftp` loadable module.

4.3.3 HTTP DMC

`Protocol name: http, https, httpg (not implemented yet)`

This DMC implements access to a data server using HTTP over TCP or over TLS protocol. It uses GET for reading and PUT for storing files. Data is always transferred in chunks due to current limitation of HTTP MCC. Listing is also supported. For that DMC extracts references from the `A` tags of the obtained HTML content. If retrieved content is not HTML or if content of its `TITLE` tag does not start from ""Index of /"" then only information about resquested URL itself is listed.

Name of plugin is `http` and it is located in {`lib`}`dmchttp` loadable module.

4.3.4 LDAP DMC

`Protocol name: ldap`

The LDAP DMC implements access to a data accessible through LDAP protocol. Currently it can only retrieve content of LDAP tree. Retrieved content is converted into XML. The DMC supports following kind of URL:

`ldap://host[:port]/base[?[attributes][?[scope][?filter]]]`

- `attributes` is comma separated list of attributes which has to be retrieved,
- `scope` is either `base`, `one` or `sub` as defined by LDAP,
- `filter` is LDAP filter.

Name of plugin is `ldap` and it is located in {`lib`}`dmcldap` loadable module.

4.3.5 LFC DMC

`Protocol name: lfc`

The LFC DMC provides an access to LCG File Catalog service and supports file listing, URL resolution and (un)registration of file locations. It uses LFC protocol implementation provided by gLite middleware.

Name of plugin is `lfc` and it is located in {`lib`}`dmclfc` loadable module.

4.3.6 RLS DMC

`Protocol name: rls`

The RLS DMC provides an access to Replica Location Service and supports file listing, URL resolution and (un)registration of file locations. It uses RLS protocol implementation provided by Globus Toolkit.

Name of plugin is `rls` and it is located in {`lib`}`dmcrls` loadable module.

Chapter 5

Future work

Interfaces in HED are mature and stable with only the security infrastructure still being extended.

Most of the work has shifted to the development of higher level libraries and services based on the HED. But the HED also continiously being expanded by adding new plugins.

There are also significant pieces of functionality not addressed in the HED yet like user-friendly configuration, efficient inter-service internal Message routing, etc.

Chapter 6

Appendices

6.1 Step-by-step instructions to add registration ability to HED services

6.1.1 General knowledge

The HED has a brand new self-register ability. If your Service is prepared to co-operate with this mechanism then there will be an InfoRegister connecting to it. There are other, so called InfoRegistrar objects present that are the HED self-register mechanism's active elements communicating with different ISIS clouds. (The InfoRegisterContainter is an instance that connects them, but this is just a good-to-know detail.) The system administrator can connect the Services with ISIS clouds connecting InfoRegisters with InfoRegistrars. The configuration (for example during testing your service) has to cover the both side. (It is possible to connect one InfoRegister to more InfoRegistrars and vice versa. See figure 6.1.)

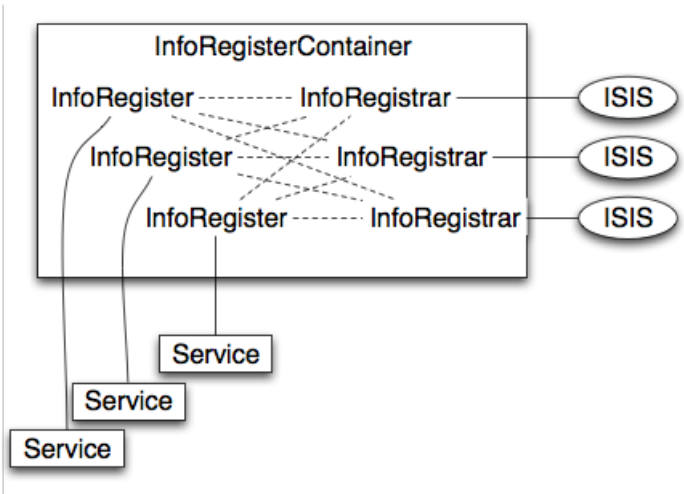

Figure 6.1: The HED internal registration infrastructure

The configuration details will be shown on the section Modify your configuration6.1.4.

The InfoRegistrar that are connected with your Service (by configuration) will periodically ask your Service for a Registration Entry what are stored in the information system. This "poll" is done by executing the RegistrationCollector function that provide an XML document in a given format. For further details see section Compose the Registration Entry6.1.3! Actually this interface can be accessed in every C++ or Python service.

The InfoRegistrar collect the simultaneous generated Registration Entries and compose a Registration Message by aggregating them (if any aggregation is possible).

The general "sandbox" echo services are already prepared to this functionality and will be referenced at the proper locations.

6.1.2 Change your source

C++ source

- Your Service have to extend the RegisteredService instead of the Service class. This class is implemented in the infosys library and not in the message library as before. (Of course the new library have also to be linked to your Service.) In your constructor please call the RegisteredService class's constructor!
- You have also to implement the RegistrationCollector function that will provide the Registration Entry XML document.

Your_Service.h

```
...
#include <arc/infosys/RegisteredService.h>
...
class Your_Service: public Arc::RegisteredService {
...
    bool RegistrationCollector(Arc::XMLNode &doc);
...
}
```

For example see: src/tests/echo/echo.h in the source tree. **Your_Service.cpp**

```
...
Your_Service::Your_Service(Arc::Config *cfg):RegisteredService(cfg) {
...
    bool Your_Service::RegistrationCollector(Arc::XMLNode &doc) {
        Arc::NS isis_ns; isis_ns["isis"] = "http://www.nordugrid.org/schemas/isis/2008/08";
        Arc::XMLNode regentry(isis_ns, "RegEntry");
        regentry.NewChild("SrcAdv").NewChild("Type") = "Your_Service_Type";
        regentry.New(doc);
        return true;
    }
...
}
```

For example see: src/tests/echo/echo.cpp in the source tree. **Makefile.am**

```
libyourservice_la_LIBADD = ... \
                           $(top_srcdir)/src/hed/libs/infosys/libinfosys.la
```

For example see: src/tests/echo/Makefile.am in the source tree.

Python source

- The PythonService is already extending the RegisteredService class so you have nothing to do on this place.
- Your only to-do is to implement the RegistrationCollector function and provide the Registration Entry XML document.

Your_Service.py

```
def RegistrationCollector(self, doc):
    regentry = arc.XMLNode('<RegEntry />')
    regentry.NewChild('SrcAdv').NewChild('Type').Set('Your_Service_Type')
    #Place the document into the doc attribute
    doc.Replace(regentry)
    return True
```

For example see: src/services/echo_python/EchoService.py in the source tree.

6.1.3 Compose the Registration Entry

The Registration Entry is an XML document with a given format. The HED internal mechanism tries to aggregate these messages in the Registration Messages.

There are 6 mandatory element in a Registration Entry. At least these should be provided by the service developer. The Type, Endponint reference and the ID of the Service, and the Expiration and Generation time of the message. If the ID is not present then it will be deputized with the Endpoint reference. Finally the Generation time of the message will be also automatically filled if missing from the Registration Entry. (See figure 6.2.) The schema of the Registration Entry:

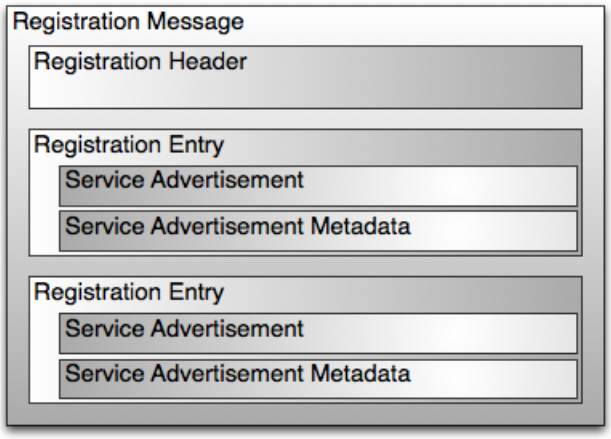

Figure 6.2: The structure of an aggregated Registration Message

```
<!-- List of the service types will be provided by the GLUE-2.0 -->
<xsd:simpleType name="ServiceTypeType">
    <xsd:restriction base="xsd:string">
    </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="NameValuePairType">
    <xsd:sequence>
        <xsd:element name="Name" type="xsd:string"/>
        <xsd:element name="Value" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ServiceAdvertisementMetadataType">
    <xsd:sequence>
        <!-- Globally unique and persistent ID of the service -->
        <xsd:element name="ServiceID" type="xsd:string" minOccurs="1"
        maxOccurs="1"/>
        <!-- Time of information generation or collection -->
        <xsd:element name="GenTime" type="xsd:dateTime" minOccurs="1"
```

```
            maxOccurs="1"/>
            <xsd:element name="Expiration" type="xsd:duration"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ServiceAdvertisementType">
        <xsd:sequence>
            <!-- General part of the Service Advertisment -->
            <xsd:element name="Type" type="isis:ServiceTypeType"/>
            <xsd:element name="EPR" type="wsa:EndpointReferenceType"/>
            <!-- Service specific part of the Service Advertisment -->
            <xsd:element name="SSPair" type="isis:NameValuePairType" minOccurs="0"
            maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="RegistrationEntryType">
        <xsd:sequence>
            <xsd:element name="SrcAdv" type="isis:ServiceAdvertisementType"
            minOccurs="1" maxOccurs="1"/>
            <xsd:element name="MetaSrcAdv" type="isis:ServiceAdvertisementMetadataType"
            minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
```

It can be also found in the source tree on the path: src/services/isis/schema/isis.xsd An examle Registration Entry XML document:

```
<RegEntry>
  <MetaSrcAdv>
    <ServiceID>http://your.domain.com/echo<ServiceID>
    <GenTime>1994-11-05T13:15:30Z</GenTime>
    <Expiration>PT3H</Expiration>
  </MetaSrcAdv>
  <SrcAdv>
    <Type>org.nordugrid.tests.echo</Type>
    <EPR>
      <Address>http://your.domain.com/echo</Address>
    </EPR>
  </SrcAdv>
</RegEntry>
```

The service type follows the GLUE2 naming convention and organize the services into categories based on their functionality. The following service types are already defined:

1. Storage:
   - org.nordugrid.storage.ahash
   - org.nordugrid.storage.bartender
   - org.nordugrid.storage.librarian
   - org.nordugrid.storage.shepherd
   - org.nordugrid.storage.hopi
2. Security:
   - org.nordugrid.security.charon
   - org.nordugrid.security.saml

- org.nordugrid.security.slcs
- org.nordugrid.security.delegation

3. Infosys:
   - org.nordugrid.infosys.isis
   - org.nordugrid.infosys.eils
   - org.nordugrid.infosys.rte-catalog
4. Execution:
   - org.nordugrid.execution.arex
   - org.nordugrid.execution.janitor
   - org.nordugrid.execution.sched
   - org.nordugrid.execution.paul
5. Accounting:
   - org.nordugrid.accounting.mars
6. Tests:
   - org.nordugrid.tests.echo
   - org.nordugrid.tests.echo_java
   - org.nordugrid.tests.echo_python
   - org.nordugrid.tests.isistest

6.1.4 Modify your configuration

The configuration have to contain the connection between the InfoRegisters (Services) and InfoRegistrars (ISIS clouds). The schema of the configuration can be also found in the source tree (src/hed/libs/infosys/InfoRegisterConfig.xsd). There is also an example configuration between the configuration templates in the source tree (src/hed/profiles/SecureP2PIIS/SecureP2PIIS.xml).

The Service configuration should contain the InfoRegister and implicitly the InfoRegistrar configuration.

```
<infosys:InfoRegister>
     <infosys:Period>PT20S</infosys:Period>
     <infosys:Endpoint>https://localhost:50000/example_service</infosys:Endpoint>
     <infosys:Expiration>PT100S</infosys:Expiration>
     <infosys:Registrar>
         <infosys:URL>some_url</infosys:URL>
         <infosys:KeyPath>some_local_path</infosys:KeyPath>
         <infosys:CertificatePath>some_local_path</infosys:CertificatePath>
         <infosys:CACertificatesDir>some_local_path</infosys:CACertificatesDir>
     </infosys:Registrar>
     <infosys:Registrar>
         <infosys:URL>some_other_url</infosys:URL>
         <infosys:KeyPath>some_local_path</infosys:KeyPath>
         <infosys:CertificatePath>some_local_path</infosys:CertificatePath>
         <infosys:CACertificatesDir>some_local_path</infosys:CACertificatesDir>
     </infosys:Registrar>
 </infosys:InfoRegister>
```

If the same InfoRegistrar would exist at more then one Serivce (the URL have to be the same) then the internal HED mechanism detect the similarity and tries to aggregate the messages if it's possible. You can also override the default Period, Endpoint, Expiration values inside the single Registrar elements and them too by written some value in the document in your source.

If the service configuration contains a

```
<NoRegister />
```

element then it won't be registered.