NORDUGRID-TECH-19
9/9/2008

The Hosting Environment of the Advanced Resource Connector middleware

J. Jönemo*, et al

*authors@address

Contents

1 Introduction **5**

2 How to read **7**

3 Architecture **9**

3.1 Requirements . . . 9

3.2 Technical design . . . 9

3.2.1 MCC . . . 10

3.2.2 Services and Clients . . . 12

3.2.3 Plexer . . . 12

3.2.4 Error handling . . . 12

3.2.5 Instantiation of the Chain . . . 13

3.2.6 Sessions and Contexts . . . 13

3.2.7 DMC . . . 13

3.2.8 Alternative implementation languages . . . 14

4 Implemented elements **15**

4.1 Implemented MCCs . . . 15

4.1.1 TCP MCC . . . 15

4.1.2 TLS MCC . . . 16

4.1.3 HTTP MCC . . . 17

4.1.4 SOAP MCC . . . 18

4.2 Implemented Security Handlers . . . 18

4.2.1 Simple List Auth . . . 18

4.2.2 Username Token . . . 18

4.2.3 X.509 Token . . . 18

4.3 Implemented DMCs . . . 19

4.3.1 File DMC . . . 19

4.3.2 GridFTP/FTP DMC . . . 19

4.3.3 HTTP DMC . . . 19

4.3.4 LDAP DMC . . . 19

4.3.5 LFC DMC . . . 19

4.3.6 RLS DMC . . . 19

5 Conclusion **21**

Chapter 1

Introduction

The Hosting Environment Daemon (HED) is the container of all the functional components of the new 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 only as much as needed. This is why the HED mostly consists of pluggable modules with some glue between them.

Because in it's 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 of numerous implementations. But 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 puts 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 the 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 that is found to be 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 suggestions how to improve architecture and code of the HED. Statements like "you are doing crap, I'll better use Axis" are understood but not welcome and are usually reacted to adequately.

Chapter 2

How to read

This document does not (yet?) include in depth description of C++ classes which constitute the HED. Instead most section contain notes entitled "Relevant classes". Technical description of those classes can be found in automatically generated "Hosting Environment (Daemon) Reference Manual" document [1].

For examples please see source code in repository [2]. 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 needed to be object oriented, efficient and provide easy access to system functionality. This eventually lead to the adoption of C++. but languages such as Java and Python were also considered at an early stage.

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 of course depend on many external libraries and components but each such dependenscy has been introduced only after due consideration.

Conservation of resources was an important goal. The present design enables many services sharing both the same process and the same network ports or even port while at the same time exhibiting a remarkably low memory footprint.

3.2 Technical design

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

The HED itself means three things:

1. the daemon (called `arched`) which hooks up the system and initialize components the way as it is described by 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 developes 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 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 interconection 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 a 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 next MCC in the chain. For information how messages are handled and about memory management policies please see API description of MCC class in [1].

The developer who writes an MCC is free to choose any 3rd party library and component to implement the functionalty 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 HTTPD service component. For that purpose ¡next¿ elements in MCC configuraton may have optional `id` attribute which allows to assign labels to all chain links to next MCCs in the chain. Supported labels are MCC dependant. 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 have 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 incomming 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 same plugin module.

Payload

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

Main content of the information is transfered 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 it's 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 it's implementation PayloadStream. It covers case of sequentially accessible information. The main purpose of that Payload is to serve protocols which define continous 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 stings. Each key may have multiple corresponding values. All pairs are handled by MessageAttributes class. Codewise there are no limitations put 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" [3].

Security Attributes are storing various aspects of Message useful for authorization decisions to be made. Those normally include 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 Secuirity 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 if Secuirity 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 way collected information may be turned into useful format. For that each Security Attribute value implements method Export for converting internal information into one of supported formats. Currently only implemented is ARC Authorization Request/Policy (see below). Please see API description of SecAttr and MessageAuth classes in [1] 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 Message as only argument. Different queues are executed at different times. Which queue is executed at which case depends on 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 as argument the Message 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 HED.

3.2.2 Services and Clients

The services are dynamically loaded on start up just like the MCCs. They almost identic 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 task of writing clients there is library `arcclient` provided which wraps task of creating Chains and Messages for widely used SOAP, HTTP, TLS, TCP communications.

3.2.3 Plexer

`Relevant classes: Arc::Plexer`

In general case multiple Services living in the HED so the incomming Message should route to the proper Service. The Plexer MCC does this job. It takes the the ENDPOINT attributes of the message collected by other MCCs compares this attribute to regular expression defined in the configuration file and forward the message to the all matching service. It acts as a dispacher. The Plexer is also special in a sence that it is not a plugin but part of the `arcloader` library.

This 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 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 advisable to convert MCC_Status error obtained from next MCC in the Chain into protocol psecific 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 configuration Loader creates single object of corresponding class. On Loader destruction all handled components are destroyed too.

3.2.6 Sessions and Contexts

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

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 it self 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 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 MessageContextElement class. The Security Attributes can also be stored in dedicated container - MessageAuthContext - with Session lifetime. At end of the Session all associated objects are destroyed.
3. The Chain lifetime - is time period while componnets making the Chain exist. This lifetime is represented by ChainContext class. Differently from other context objects this one doe not allow free manipulation of contained objects. Instead it provides an interface to some internal structures of Loader object. Those include factories and lists of object 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 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

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 location of data directly - like HTTP, FTP, LDAP - the DMC can be used with indirect/indexing endpoints. Those are endpoints which define only 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 [4].

More in depth techical information about the DMC can be found in [5].

3.2.8 Alternative implementation languages

In order to facilitate the developement of services, API bindings for languages other than C++ are provided and some service developement has already been done in Python language. Currently only available language bindings are Python and Java. And 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 infractructure itself. Although strictly not belonging to infrastructure this minimal set of components is necessary to make inrastructure useful.

4.1 Implemented MCCs

4.1.1 TCP MCC

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

`Library name: {lib}mcctcp`

`SEcuruty handler queues: incoming, outgoing`

The server side TCP MCC in the HED is special in that it produces messages by listening on a network socket rather than passing on 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 configuered 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 next MCC whihc can be used for communicationg through open TCP channel. Currently it ignores any payload attached to the returned Message.

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 transfered ober TCP connection. Accepted Payload type of incoming Message is `PayloadRawInterface`. Returned Payload is of `PayloadStreamInterface` type. It represents established TCP conenction and `may` be used by previous MCCs in chain for direct communication. It is still prefered 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

`Securuty handler queues:` `incoming, outgoing`

The server and client TLS MCCs provide transport level security (TLS) over any stream channel. Curently 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 next MCC. This object is maintained inside Message Context under name `tls.service` and is destroyed when Context becomes inactive. Currenlty this MCC does not expect any payload to be returned from rest of the chain and passes no payload to previous MCC.

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

Both client ans server side MCCs are implemented using OpenSSL toolkit [6], use X.509 infrastructure [7]

for establishing secure connection and may be configured to get private key, certfificate or proxy credentials from files esiding at local file system. It also possible to specify location of Certification Authority cerfificate 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`

`Securuty handler queues: incoming, outgoing`

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 next MCC as `PayloadRawInterface` payload. In response this MCC expects also the Message with `PayloadRawInterface`. It is then prepended with HTTP reponse header and pushes it into initially providded stream channel. for output it returns empty `PayloadRawInterface` payload.

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

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

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`

`Securuty handler queues: incoming, outgoing`

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

4.2 Implemented Security Handlers

4.2.1 Simple List Auth

`Plugin name: arc.authz`

`Library name: {lib}arcpdc`

4.2.2 Username Token

`Plugin name: usernametoken.handler`

`Library name: {lib}arcpdc`

4.2.3 X.509 Token

`Plugin name: x509token.handler`

```
Library name: {lib}arcpdc
```

4.3 Implemented DMCs

4.3.1 File DMC

4.3.2 GridFTP/FTP DMC

4.3.3 HTTP DMC

4.3.4 LDAP DMC

4.3.5 LFC DMC

4.3.6 RLS DMC

Chapter 5

Conclusion

Bibliography

[1] "Hosting Environment (Daemon) Reference Manual," Autogenerated Doxygen document. [Online]. Available: http://svn.nordugrid.org/trac/nordugrid/export/9917/arc1/trunk/doc/ARC%1-API.pdf

[2] "NorduGrid subversion - ARC1," Web site. [Online]. Available: http://svn.nordugrid.org/trac/nordugrid/browser/arc1

[3] W. Qiang *et al.*, *Security Framework of ARC1*, The NorduGrid Collaboration, NORDUGRID-TECH-16. [Online]. Available: http://svn.nordugrid.org/trac/nordugrid/export/9917/arc1/trunk/doc/sec%/SecurityFrameworkofARC1.pdf

[4] A. Konstantinov, *Protocols, Uniform Resource Locators (URL) and Extensions Supported in ARC*, The NorduGrid Collaboration, NORDUGRID-TECH-7. [Online]. Available: http://www.nordugrid.org/documents/URLs.pdf

[5] M. Ellert, *ARC Data Manager Component (DMC)*, The NorduGrid Collaboration, NORDUGRID-TECH-16. [Online]. Available: http://svn.nordugrid.org/trac/nordugrid/export/9917/arc1/trunk/doc/dmc%/dmc.pdf

[6] "The Open Source toolkit for SSL/TLS," Web site. [Online]. Available: http://www.openssl.org/

[7] "Public-Key Infrastructure (X.509) (PKI), Proxy Certificate Profile." [Online]. Available: http://rfc.net/rfc3820.html