NORDUGRID-MANUAL-9
14/5/2009

Web Service programming Tutorial

This document is still under development, but don't hesitate sending your comments and suggestions to glodek@inb.uni-luebeck.de.

Michael Glodek*, Steffen Möller

*glodek@inb.uni-luebeck.de

Contents

1 Introduction

This document gently introduces to the preparation of standalone Web Services and cognated clients with the Advance Resource Connector (ARC). The reader is guided through a series of practically oriented examples. Each is accompanied by explanations of the main concepts of the software architecture. No particular skills are required to follow the presented steps. It is however beneficial to have some basic comprehension of the programming language C++, the file format XML and any regular UNIX shell. To start, the user shall have an installation of ARC-1, as it is performed with current RPM or .deb packages*.

The ARC server provides access to services. These are executed on the same machine that the server is installed on. Some services, like the A-REX service, will delegate computational efforts to other machines via additional software like a local queueing system. But the A-REX service itself does not migrate away to other machines - at least this has not been implemented yet. Services may comprise computational tasks (e.g., some secret and strongly patented algorithm), allow arbitrary computations (like ARC's A-REX or Amazon's clouds) or provide access to other resources like a particular database or to arbitrary disk space (like ARC's hopi or bartender).

Services are invoked using software programs, which are referred to as clients. These programs may directly perform the interactions of human users or request a performance of other services of another server (here the first service is in the role of a client, too). The tasks of servers and clients are well defined. If servers are not busy reacting to a client's request, then they are waiting. The task of servers can be subdivided into:

- Wait for client request
- Receive the request
- Perform the desired service
- Create a response for the client
- Send response to the client
- Wait for next client request

While the server waits passively for a request, the client acts actively:

- Create a request
- Transmit the request to the server which provides the desired service.
- Wait for the reply
- Receive the response from the server.

Servers provide access to a single or a set of services. In ARC services are provided by the Hosting Environment Demon (HED) which enables the installation of several services on one network access point.

In general, the implementation of a client is easier than the implementation of the server. ARC is designed as a middleware which encapsulates typical challenges of server-client infrastructures (security, exception handling, extensibility) and abstracts from the underlying computer architecture (heterogeneity of computers, computer location, protocols). The core of ARC is the prior mentioned HED (Hosting Environment Daemon). To the administrator HED is mostly visible as the single binary that is started and configured by a file that describes services' that shall be prepared by HED. Conceptionally, the HED determines how services shall be organised both as a principle, and for any given ARC-run server. HED determines the very explicit presence or absence of a service at a particular address. Its many configurable layers are described later in this tutorial. At the time of writing, the endpoint of the HED will be a SOAP based Web Service.

*see http://wiki.nordugrid.org for explicit instructions for setup

SOAP is a long established XML-based standard for Web Service communication. ARC abstracts from this format, but nevertheless it is useful to be aware of the underlying message transport mechanism.

1.1 Hosting Environment Daemon

The HED which was already mentioned in the section above is an essential part of the ARC middleware. Its task is to provide the hosting of various services at the application level and is based on the idea modularity. The HED consists of components called Message Chain Components (MCC), Plexer, Services, and several modules, which shall aid the programmer to simplify the development of the Web Services: Config, Loader, Logging, XMLNode et cetera. The MCCs are ordered within a layered structure. Each MCC provides a certain functionality such as communication between two applications (MCC TCP), secure data transfer (MCC TLS) or client-server architecture (MCC HTTP). Figure 1.1 illustrates a typical setup of a server-sided HED.

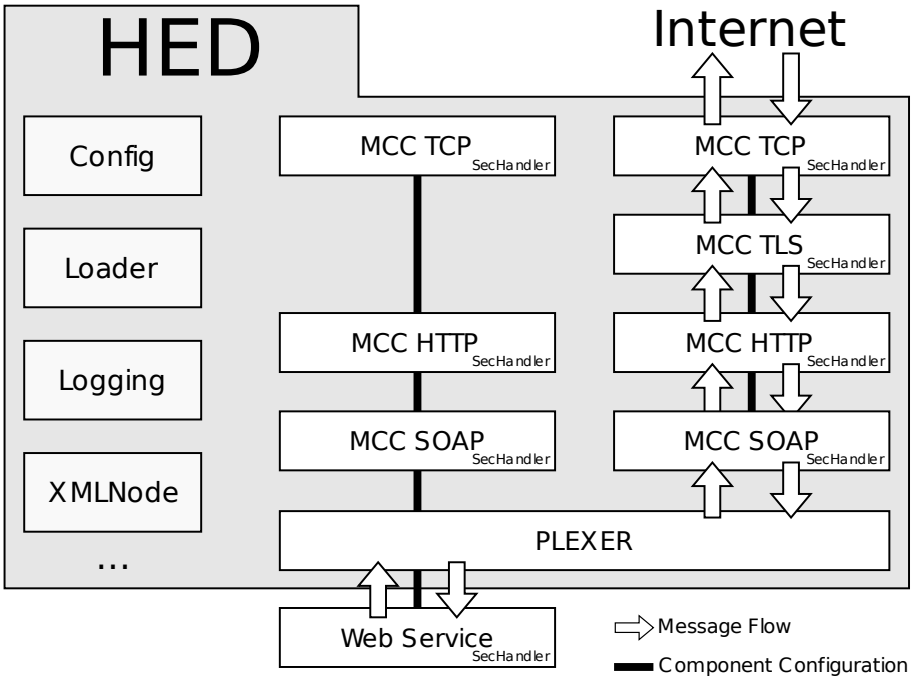


Figure 1.1: Example of a HED structure providing a Web Service The HED consists of several layers which are connected with each other. The arrow indicate the message flow of the used configuration. The MCCs and Services may hold a SecHandler in order to give additional security. Additional modules (on the left) are used within the HED.

The MCCs are connected which each other and passing incoming and outgoing message to the upper or the respective lower layer. In case of outgoing messages a MCC is wrapping the data of the upper layer as a payload into their own protocol (while incomming message will be unwrapped). The lowest layer has to be a transport protocol like TCP which enables the transmission between two applications. Technically there are even lower layers required needed for the communication but they are already realised within the operating system and the network card. TCP is well suited for the application because it provides a reliable and ordered delivery of a byte stream. Above the TCP layer is the Hypertext Transfer Protocol (HTTP). In case a security layer is desired (i.e. for authentication), the TLS (Transport Layer Security formerly known as SSL, Secure Sockets Layer) has to be inserted between the TCP and HTTP layers. The HTTP provides the client-server architecture. It is stateless but offers several extensions for requests, header information and status codes. Furthermore HTTP enables the usage of Uniform Resource Locators (URL) which is used by ARC to multiplex between different services. The multiplexing layer can also be defined within the HED and is called Plexer. Depending on the path of the URL the Plexer passes messages to a defined SOAP service which again passes the message to the Web Service [**?**].

The structure of the HED can be configured freely by modifying the server configuration file. A first impression how the HED can be configured shall be given in the following example which corresponds to the HED shown in Figure 1.1. It uses the ARC echo service that is shipped for testing purposes with the ARC source code.

1.1.1 The ARC Echo Web Service

The first example shall give a basic understanding in how to configure the HED for setting up a novel service. The task is to setup the regular ARC Echo Web Service. On the command line, all that needs to be done is to invoke the *arched* daemon with a suitable server configuration file. Such a fitting server configuration file is shown in Listing 1.1. It is written in XML. Its structure is based on the XSD schema listed in 7.2 in Listing 7.1.

Listing 1.1: HED configuration file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ArcConfig xmlns="http://www.nordugrid.org/schemas/ArcConfig/2007" xmlns:tcp="http://www.
       nordugrid.org/schemas/ArcMCCTCP/2007"  xmlns:echo="urn:echo_config">
3    <!-- Configuration of the server entities -->
4    <Server>
5      <Pidfile>/var/run/arched.pid</Pidfile>
6      <Logger level="VERBOSE">/var/log/arched.log</Logger>
7    </Server>
8    <!-- Path to the module libraries -->
9    <ModuleManager>
10     <Path>/usr/lib/arc/</Path>
11   </ModuleManager>
12   <!-- Plugins needed to run the server -->
13   <Plugins><Name>mcctcp</Name></Plugins>
14   <Plugins><Name>mcctls</Name></Plugins>
15   <Plugins><Name>mcchttp</Name></Plugins>
16   <Plugins><Name>mccsoap</Name></Plugins>
17   <Plugins><Name>echo</Name></Plugins>
18   <!-- Structure of the Message Component Chain -->
19   <Chain>
20     <Component name="tcp.service" id="tcp">
21       <next id="http"/>
22       <tcp:Listen><tcp:Port>60000</tcp:Port></tcp:Listen>
23     </Component>
24     <Component name="http.service" id="http">
25       <next id="soap">POST</next>
26     </Component>
27     <Component name="soap.service" id="soap">
28       <next id="plexer"/>
29     </Component>
30     <Plexer name="plexer.service" id="plexer">
31       <next id="echo">^/Echo$</next>
32     </Plexer>
33     <Service name="echo" id="echo">
34       <echo:prefix>[ </echo:prefix>
35       <echo:suffix> ]</echo:suffix>
36     </Service>
37   </Chain>
38 </ArcConfig>
```

The first line of the XML file contains the XML declaration. Several attributes can be defined here but at least the XML version should be specified. The configuration as a whole is encapsulated by the element *ArcConfig*. It also performs the setting of namespaces, which are expected to be mostly invariant across all server installations. The *Server* element, to be found at line 4, provides basic settings for the daemon such as the location of the Pid-file or ofthe Log-file. The *ModuleManager* holds the *path* to the plugin libraries. Several paths may be defined here. Due to the fact that the first library matching the plugin name is loaded, the order is relevant. The path should be at least assigned to the ARC installation directory. Otherwise, the set of ARC MCC plugins might not be found. The next elements are holding the name of the plugins to be loaded (line 13). The names have to correspond to the names of the dynamic libraries within the path defined in the *ModuleManager*. Finally, the chain, which creates the layered structure, is declared within the element *Chain*. It is composed of the elements *Component*, *Plexer* and *Service*. Due to a MCC plugin may contain several components, one has to specify more exact which one to load. This is to be done using the *name* attribute i.e. the plugin *mcchttp* realises two components: *http.service* and *http.client*. Since we

are defining a configuration file for a service, the *http.service* should be used. Furthermore the elements may have the attribute *id* as a unique identifier within the file. Regarding the components and the services a plugin (implemented as a *.so file that is searched in the paths previously specified) has to be loaded which provides the implementation of its functionality. The further configuration depends on the plugin's individual functionality.

A *Chain* will determine the path that an event takes to be noticed. The first component of the chain is the *tcp.service*. As to be seen, the port 60000 is assigned to the server to listen to. The port can be an arbitrary number. However, the number 60000 is commonly used for a HED providing the A-REX Web Service (the one organising the computation on a site) and the number 50000 is found commonly used for a HED running Web Services for storage. This way, one can organise a single server to offer multiple HED instances. The stream which is received by this component is passed to the component defined by the *next* attribute. In the present case, it will be passed to the component with the *id* attribute named *http*. In this way, the message is be passed to higher layers until it reaches the *Plexer*. Depending on the path of the URL which was demanded by the HTTP, the Plexer multiplexes the request to the right Web Service The plexer will thus allow to have multiple Web Service listen to the same port. The path, declared within the *next* attribute, corresponds to a regular expression and leads to the service with the id named *echo*. Hence, the service will now be reachable under URL *http://localhost:60000/echo*. It will finally process the message and create a new message which will be returned all the way back to the TCP component. The Echo Web Service itself is provided by the ARC package. Additional invariant parameters are passed by the elements *prefix* and *suffix*. They set the characters which will enclose the echoed string.

To create the daemon corresponding to the defined server configuration file, ARC provides the program called *arched*. The manner of invokation is to been seen in Listing 1.2.

Listing 1.2: Invokation of the Arched Daemon.

```
$ arched -c arcecho_no_ssl.xml && echo Daemon started || echo Daemon failed
 Daemon started
$
```

The name of the program is followed by the parameter *-c*. The string following that argument contains the path to the configuration file. Once the server is running, the client can be used as described in Listing 1.3.

Listing 1.3: Usage of the Arc echo client.

```
$ arcecho http://localhost:60000/Echo text
[ text ]
```

In this example, the server and the client are running on the same computer such that the hostname *localhost* can be utilized. For debugging it is advisable to check the logfile which was assigned inside the server configuration file. It is commonly found at /var/log/arched.log.

The following chapters explain how to develop custom Web Services. The first one will be the simple Time Web Service.

2 The Time Web Service

The first implementation of a Web Service will be a simple time service. When the service receives a request of a client, it will respond with a string containing the system time. For simplicity, the service will not parse the request but always return the current time. It is therefore easier than the example of an Echo Web Service which additionally has to extract the incoming message in order to get the string which shall be echoed. In the following subsections the implementation of the service and the client will be presented.

The source code of the examples given in this tutorial are available in the directory *src*[*] that accompanies the electronic version of this document.

2.1 Service

In ARC, the services are implementated as plugins such that they can easily be included or excluded by modifiying the server configuration file as it was seen in Listing 1.1. The class needed to be implemented for the Web Service inherits from the class *Service*, which itself is defined in the ARC library. The proper class definition of the object *TimeService* is defined in the corresponding header as shown in Listing 2.1.

Listing 2.1: Header file of the Time Web Service.

```
1  /**
2   * Header file of Time Web Service
3   *
4   * The task of the service is alway to reply with the
5   * current system time.
6   */
7
8  #ifndef __TIMESERVICE_H__
9  #define __TIMESERVICE_H__
10
11 #include <arc/message/Service.h>
12 #include <arc/Logger.h>
13
14 namespace ArcService
15 {
16
17 class TimeService: public Arc::Service
18 {
19     protected:
20
21         /**
22         *  Arc-intern logger. Redirects the log messages into the
23         *  file specified within the server configuration file.
24         */
25         Arc::Logger logger;
26
27         /**
28         * XML namespace of the outgoing SOAP message.
29         */
30         Arc::NS ns_;
31
32     public:
33
34         /**
35         * Constructor of the Time Web Service.
36         */
37         TimeService(Arc::Config *cfg);
38
39         /**
40         * Destructor of the Time Web Service.
41         */
42         virtual ~TimeService(void);
43
```

[*]If this was not shipped with the document you are reading, then please inspect http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/doc/ws-programming-tutorial.

```
44          /**
45          * Implementation of the virtual method defined in MCCInterface
46          * (to be found in MCC.h).
47          * @param inmsg incoming message
48          * @param inmsg outgoing message
49          * @return Status of the achieved result
50          */
51          virtual Arc::MCC_Status process(Arc::Message& inmsg, Arc::Message& outmsg);
52  };
53
54  } //namespace ArcService
55
56  #endif
```

The class *Service* renders the possibility to implement the method *process*. Listing 2.2 contains the C++ implementation of the Time Web Service plugin. The key element for each plugin is the struct *PluginDescriptor* named *PLUGINS_TABLE_NAME* to be seen in line 32. Within the struct, a unique plugin name, the type of the plugin, its version and a pointer to a function which returns the instance of the time service has to be defined. The function which returns the pointer of the instance is desclared in line 18. Within the interface an object *PluginArgument* is demanded as a parameter. But since the plugin that is now written inherits from the class *Service*, one can expect the *PluginArgument* to be of a special subclass *ServicePluginArgument*. The *ServicePluginArgument* provides access to the server configuration and to the message chain of the service which later may be useful.

The Time Web Service is implemented using the namespace *ArcService*. The constructor can be found in line 52 and the deconstructor in line 62. In the context of the simple service they don't contribute any functionality. Only the constructor is defining the XML null-namespace for the time service.

Listing 2.2: Implementation of the Time Web Service.

```
1   #include <arc/loader/Plugin.h>
2   #include <arc/message/PayloadSOAP.h>
3
4   #include <stdio.h>
5   #include <time.h>
6
7   #include "timeservice.h"
8
9   /**
10   * Within the following variable PLUGINS_TABLE_NAME a pointer to a
11   * method with a certain signature has to be stored.
12   *
13   * This method has to be capable to process the argument PluginArgument
14   * and must return a pointer to a plugin, which is actually the timeservice.
15   *
16   * Therefore the method initializes the time service and returns it.
17   */
18  static Arc::Plugin* get_service(Arc::PluginArgument* arg)
19  {
20      // The dynamic cast can handle NULL
21      Arc::ServicePluginArgument* mccarg = dynamic_cast<Arc::ServicePluginArgument*>(arg);
22      if(!mccarg) return NULL;
23
24      return new ArcService::TimeService((Arc::Config*)(*mccarg));
25  }
26
27
28  /**
29   * This PLUGINS_TABLE_NAME is defining basic entities of the implemented .
30   * service. It is used to get the correct entry point to the plugin.
31   */
32  Arc::PluginDescriptor PLUGINS_TABLE_NAME[] = {
33      {
34          "time",                 /* Unique name of plugin in scope of its kind */
35          "HED:SERVICE",          /* Type/kind of plugin */
36          1,                      /* Version of plugin (0 if not applicable) */
37          &get_service            /* Pointer to constructor function */
38      },
39      { NULL, NULL, 0, NULL }     /* The array is terminated by element */
40                                  /* with all components set to NULL*/
41  };
42
43  using namespace Arc;
44
45  namespace ArcService
46  {
47
48      /**
```

```
49      * Constructor. Calls the super constructor, initializing the logger and
50      * setting the namespace of the payload.
51      */
52      TimeService::TimeService(Arc::Config *cfg)
53              : Service(cfg),logger(Logger::rootLogger, "Time")
54      {
55          // Setting the null-namespace for the outgoing payload
56          ns_[""]="urn:time";
57      }
58
59      /**
60      * Deconstructor. Nothing to be done here.
61      */
62      TimeService::~TimeService(void)
63      {
64      }
65
66      /**
67      * Implementation of the virtual method defined in MCCInterface.h (to be found in MCC.h).
68      *
69      * This method processes the incoming message and generates an outgoing message.
70      * For this is a very simple service, the time always being returned without examining
71      * the the incoming message.
72      *
73      * @param inmsg incoming message
74      * @param inmsg outgoing message
75      * @return Status of the result achieved
76      */
77      Arc::MCC_Status TimeService::process(Arc::Message& inmsg,Arc::Message& outmsg)
78      {
79          logger.msg(Arc::DEBUG, "Timeservice has been started...");
80
81          // Get time and create an string out of it
82          time_t timer;
83          timer=time(NULL);
84          std::string loctime = asctime(localtime(&timer));
85          loctime = loctime.substr(0,loctime.length()-1);
86
87          // Create an output message with a tag time containing another
88          // tag timeResponse which again shall contain the time string
89          Arc::PayloadSOAP* outpayload = NULL;
90          outpayload = new Arc::PayloadSOAP(ns_);
91          outpayload->NewChild("time").NewChild("timeResponse")=loctime;
92
93          outmsg.Payload(outpayload);
94
95          // Create well formated, userfriendly (second argument) XML string
96          std::string xmlstring;
97          outpayload->GetDoc(xmlstring, true);
98          logger.msg(Arc::DEBUG, "Response message:\n\n%s",xmlstring.c_str());
99
100         logger.msg(Arc::DEBUG, "Timeservice done...");
101
102         return Arc::MCC_Status(Arc::STATUS_OK);
103     }
104
105 }//namespace
```

The method *process*, defined in line 77, realises the desired service. The parameters *inmsg* and *outmsg* are references to the incoming and outgoing messages. Furthermore, the method *process* returns the object *MCCStatus* that is representing the result of the service achieved. Within the method *process*, the service first determines the current time using a system library and then creates a SOAP payload using the object *PayloadSOAP* on line 89. The object *PayloadSOAP* provides enough functionality to create a XML message that is conform to the SOAP protocol. The appearance of the generated request and its response will be discussed later on page 14.

In the current case, the payload consists of two elements, one nested into the other. The element *time* holds the element *timeResponse* which again holds the string containing the current time. Within the line 93 the payload is stored in the message. When nothing unexpected happened while the message was processed, the function returns an *MCCStatus* which was instanciated with the state *OK*, as to be seen in line 102.

To run the service, it needs to be compiled into a dynamic library. Here this is named *libtimeservice.so*. The name is arbitrary but needs to be in sync with the configuration file of the HED. For the installation, the library needs to be copied to a directory that is accessible for the HED. Any would do, but most likely one will select one that is locally mounted, in order to reduce dependencies on external hardware. For his example, the installation direction was set to */tmp/arc/tutorial/lib*. It was also specified by

the *ModuleManager* of the server configuration file. Once the server is running, the paths cannot be changed.

A suitable server configuration file which engages the time service is shown in Listing 2.3. A new *Path* element has been introduced into the element *ModuleMananger* in line 7. It assigns the location of the created dynamic library. On line 16 the library is explictly mentioned to be loaded as a plugin. Another important change has been done within the *Plexer* element in line 29: a rule has been introduced which redirects the request to the time service, if the path of URL is *time*. Please note that the URL setting is case sensitive and corresponds to a regular expression.

Listing 2.3: Configuration for the Time Web Service.

```
1  <?xml version="1.0"?>
2      <ArcConfig xmlns="http://www.nordugrid.org/schemas/ArcConfig/2007" xmlns:tcp="http://www
           .nordugrid.org/schemas/ArcMCCTCP/2007" xmlns:time="urn:time_config">
3      <Server>
4          <Pidfile>/var/run/arched.pid</Pidfile>
5          <Logger level="VERBOSE">/var/log/arched.log</Logger>
6      </Server>
7      <ModuleManager>
8          <!-- Own library path. The order is important!-->
9          <Path>/tmp/arc/tutorial/lib/</Path>
10         <Path>/usr/lib/arc/</Path>
11     </ModuleManager>
12     <Plugins><Name>mcctcp</Name></Plugins>
13     <Plugins><Name>mcctls</Name></Plugins>
14     <Plugins><Name>mcchttp</Name></Plugins>
15     <Plugins><Name>mccsoap</Name></Plugins>
16     <Plugins><Name>timeservice</Name></Plugins>
17     <Chain>
18         <Component name="tcp.service" id="tcp">
19             <next id="http"/>
20             <tcp:Listen><tcp:Port>60000</tcp:Port></tcp:Listen>
21         </Component>
22         <Component name="http.service" id="http">
23             <next id="soap">POST</next>
24         </Component>
25         <Component name="soap.service" id="soap">
26             <next id="plexer"/>
27         </Component>
28         <Plexer name="plexer.service" id="plexer">
29             <next id="time">^/time$</next>
30         </Plexer>
31         <Service name="time" id="time">
32             <next id="time"/>
33         </Service>
34     </Chain>
35 </ArcConfig>
```

In order to start the service, the *arched* command, shown in Listing 2.4, has to be used. It is recommended to examine the log file */var/log/arched.log* if the server configuration file has been modified to assert the start was successful.

Listing 2.4: Invokation if the arched daemon containing the Time Web Service

```
$ arched -c arched_timeservice.xml && echo Daemon started || echo Daemon start failed
Daemon started
```

2.2 Client

Up to now we have implemented a server program which is running on a particular machine and waiting for other programs to ask for the time. What is still needed is the client to send the request to the server and interprete the response.

This section is introducing the implementation of the client. Its main task is to interprete the returned message. In order to be able to communicate with the service providing daemon, the client needs to use an

identical protocol stack. Likewise to the HED, it will also be created dynamically using the MCCs. But as one will see, the creation of the protocol stack on client side is very simple.

The source code of the client can be inspected in Listing 2.5 and consists basically of one main function. Within the line 15 the object *Logger* is initialized. Due to the configuration of the the *Logger* in the following lines, the messages passed to it will be redirected to the standard error stream. For the components of the protocol stack, which are likewise loaded dynamically, one needs to specify the installation directory of ARC in line 21.

Listing 2.5: Souce code of the client program

```
1  #include <arc/ArcLocation.h>
2  #include <arc/message/MCC.h>
3  #include <arc/client/ClientInterface.h>
4  #include <arc/client/UserConfig.h>
5
6  #include <iostream>
7  #include <string>
8
9  using namespace Arc;
10
11 int main(int argc, char** argv)
12 {
13
14     // Initiate the Logger and set it to the standard error stream
15     Arc::Logger logger(Arc::Logger::getRootLogger(), "time");
16     Arc::LogStream logcerr(std::cerr);
17     Arc::Logger::getRootLogger().addDestination(logcerr);
18     Arc::Logger::rootLogger.setThreshold(Arc::WARNING);
19
20     // Set the ARC installation directory
21     Arc::ArcLocation::Init("/usr/lib/arc");
22     setlocale(LC_ALL, "");
23
24     // URL to the endpoint
25     std::string urlstring("http://localhost:60000/time");
26     printf("URL: %s\n",urlstring.c_str());
27
28
29     // Load user configuration (default ~/.arc/client.xml)
30     std::string conffile;
31     Arc::UserConfig usercfg(conffile);
32     if (!usercfg) {
33         printf("Failed configuration initialization\n");
34         return 1;
35     }
36
37     // Basic MCC configuration
38     Arc::MCCConfig cfg;
39     Arc::URL service(urlstring);
40
41     // Creates a typical SOAP client based on the
42     // MCCConfig and the service URL
43     Arc::ClientSOAP client(cfg, service);
44
45     // Defining the namespace and create a request
46     Arc::NS ns("", "urn:time");
47     Arc::PayloadSOAP request(ns);
48     request.NewChild("time").NewChild("timeRequest") = "";
49
50
51     // Process request and get response
52     Arc::PayloadSOAP *response = NULL;
53     Arc::MCC_Status status = client.process(&request, &response);
54
55     // Test for possible errors
56     if (!status) {
57         printf("Error %s",((std::string)status).c_str());
58         if (response)
59             delete response;
60         return 1;
61     }
62
63     if (!response) {
64         printf("No SOAP response");
65         return 1;
66     }
67
68
69     // Extract answer out of the XML response and print it to the standard out
70     std::string answer = (std::string)((*response)["time"]["timeResponse"]);
```

```
71      std::cout << answer << std::endl;
72
73      delete response;
74
75      return 0;
76  }
```

The URL of the service is defined in line 25. It fits to the server configuration file which was shown in Listing 2.3. The daemon is listing on port 60000 and is running on the localhost. The service itself uses HTTP and can be reached by using the path *time*. Line 30 loads the standard configuration file, which is usually almost empty. As a result, no additional security procedures will later be linked into the client-sided stack. The protocol stack is created in line 43. Thanks to the service URL the information for the configuration is completed.

The constructor of the class *ClientSOAP* creates a SOAP client which uses HTTP and addresses the port 60000 on localhost. If HTTPS would have been denoted within the URL, the stack would have been extended by an additional TLS protocol. Once the client stack is ready, the request to the service can be created. This is done in line 46. Even the appearance of the request is unimportant to the service, the message is composed of two convoluted elements *time* and *timeReqest*. The request will be processed in the next line by the method *process* of the object *client*. The object of the class *MCCStatus* which is returned by the method indicates the success of the processing. After verifying the response, the message will be passed to the standard out in line 70.

The client program has to be compilied to an ordinary binary such that it can directly be executed from the command line. Assure that the service is running and use the client as demonstrated in Listing 2.6. The

Listing 2.6: Invokation of the time client

```
$ ./timeclient
Wed Feb 18 11:20:30 2009
```

transmitted messages are shown in the Listings 2.7 and 2.8. The enclosing element is *Envelope* which may contain an element *Header* and must contain an element *Body*. The data created by the object *PayloadSOAP* is to be found within the *Body*.

Listing 2.7: Request message created by the client

```
1 <soap-env:Envelope xmlns="urn:time" xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding
      /" xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.
      org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
2   <soap-env:Body>
3     <time>
4       <timeRequest></timeRequest>
5     </time>
6   </soap-env:Body>
7 </soap-env:Envelope>
```

Listing 2.8: Responce message of the service

```
1 <soap-env:Envelope xmlns="urn:time" xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding
      /" xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.
      org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
2   <soap-env:Body>
3     <time>
4       <timeResponse>Wed Feb 18 11:20:30 2009</timeResponse>
5     </time>
6   </soap-env:Body>
7 </soap-env:Envelope>
```

3 Web Services Description Language

3.1 Background

For Web Services, the mere communication, as for instance we know if from traditional CGI programming, is not everything. The communication is performed via the SOAP protocol, and there are several implementations of SOAP on top of CGI. What the here explained WSDL (Web Services Description Language) is about is the *talking* about Web Services. In a now standardised XML-based format, one can emply WSDL to describe both technically and for regular users the performance of the Web Service.

WSDL may be imagined as a meta language. Beside the description of the service, WSDL renders the possibility that services can be discovered by service brokers*. The discovery of services may be useful for an automatic adaption of a running client, i.e. if the endpoint of the service changes or to share the load of demands. The WSDL service specification enables a foreign programmer to implement a suitable client which interfaces the service in a proper way. WSDL describes the structure of the messages, the intended use of the messages, the supported protocols and the endpoint of the service (URL).

WSDL documents are composed of five main elements:

- **Types** — The element *types* contains the description of the message structures and is written in the language XSD (XML Schema Definition). The data types used for the messages of the Echo Web Service are defined starting with the line 14.
- **Message** — Within the element *message* a subset of the previously defined types are assigned to be a message. In case of the Echo Web Service two messages are designated: echoRequest and echoResponse, compare lines line 52 and 55.
- **Port type** — The element *portType* is used to assign the messages to an interface. Four interface types can be distinguished: One-way (input), request-response (input, output), solicit-response (input, output, fault) and notification (output). The Echo Web Service realises a solicit-response interface which is declared subsequent to line 61.
- **Binding** — The protocol and the data format used for the message transmission are denoted in the element *binding*. Possible style attributes are *document* or *rpc* (Remote Procedure Call). The transport attribute defines the protocol which is regularly HTTP. For each operation a corresponding SOAP action has to be defined which assignes a messages to be either *literal* or *encoded* [**?**]. The service implemented in this chapter will use the binding *document/literal* and the protocol HTTP which is to be seen in the lines follow line 71.
- **Service** — Within the last WSDL element *service* the endpoint of the service gets specified. As to be seen in line 85 the endpoint of the Echo Web Service is assigned to *http://localhost:60000/echo*.

To understand an WSDL document, one best starts from the Service entry, commonly located at the end. It will describe what the service is all about. From there, move upwards to learn about the operations and their parameters. Most items in the WSDL document have their own tags that provide names or descriptions.

3.2 How to prepare a WSDL document

It should be said that the preparation of WSDL documents not ultimate fun to do. And it is tricky in the sense that one does mistakes easily that are difficult to spot.

*The here referenced brokering is meant to act on a global registry of services and has little practical meaning. It is not to be confused with the brokering of grid resources, for which ARC-1 provides a web service on its own, which is a key functionality of any computational grid.

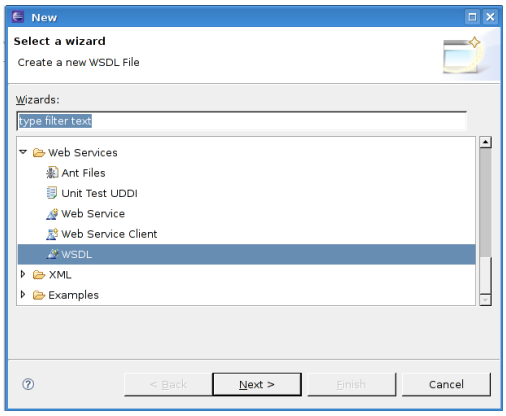


Figure 3.1: Eclipse dialog to create a new file. WSDL is selected.

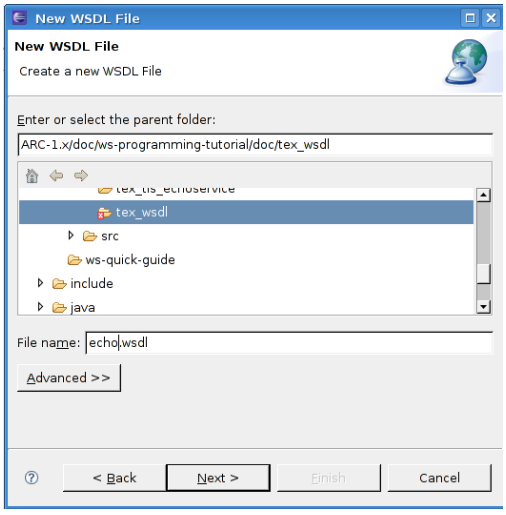


Figure 3.2: The default settings are fine for a WSDL service that communicates with ARC.

The community hence started to use computational tools to draft these documents. The most prominent of these is probably Eclipse. Since its version 3.4 (you might need to install it directly from `http://www.eclipse.org` if your distribution is a bit behind or if you are working with Windows) it offers a web tools platform (WTP) module with which WSDL files can be crafted graphically. To install it, point the url http://download.eclipse.org/webtools/updates/ to the Eclipse updater, if it is not already listed.

3.2.1 Preparation of the Echo WSDL file with Eclipse

To prepare a WSDL document with Eclipse, one should find the right file type in the typical Eclipse dialog window. This is shown in figure 3.1. The empty WSDL document is not empty, but the application starts with a minimal setup: a service that expects some input and provides some immediate return value. This is already much like the echo web service. One basically only needs to add the documentation of the service and to name the parameters. To do so, just click on the items displayed in the graphical representation and/or find respective entries in the tabs below.

3.2.2 Inspection of Web Services with Taverna

One will fine many specifications that are needed by WSDL to glue the different parts of the WSDL file together. The service references the bindings and the bindings reference the port and so on. One should not become overly nervous about it. The regular programmer does not see those internal names. A good sanity

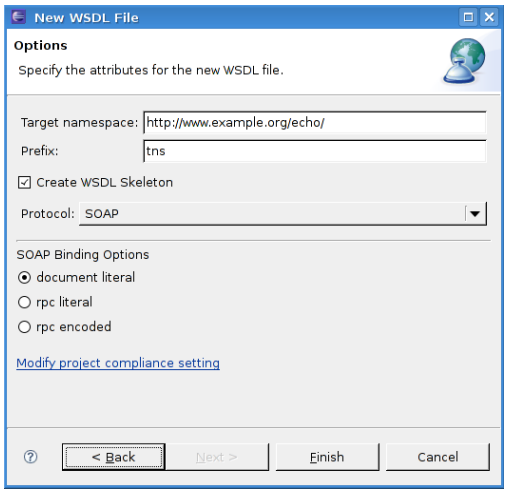


Figure 3.3: The project first shown has already a strong similarity to the Echo web service that is planned for.

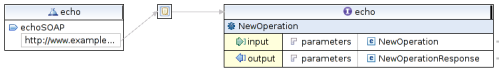


Figure 3.4: A first usable web service description is achieved solely by specifying the address and passing the arguments the known type *string*.

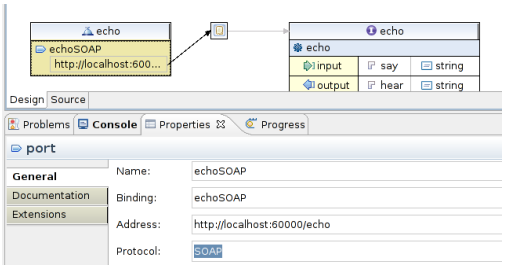


Figure 3.5: The WSDL for echo in a very simplistic manner.

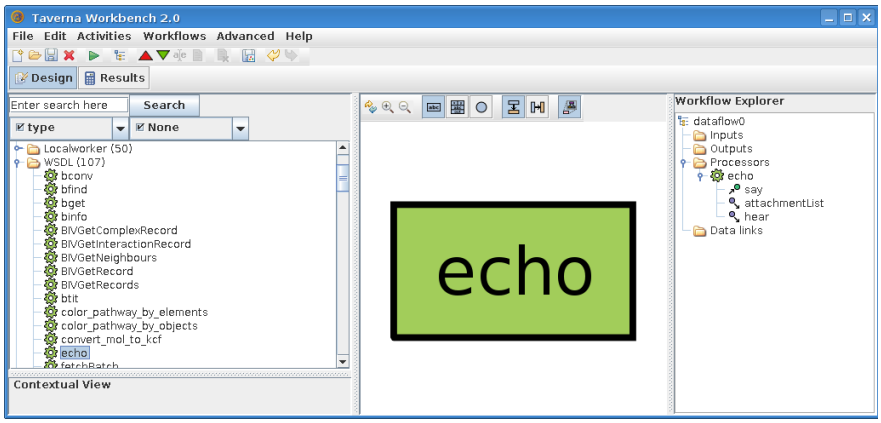


Figure 3.6: Taverna learned about the Echo Web Service and presents it. No internal names of WSDL appear, only the name of operations and paramters are shown.

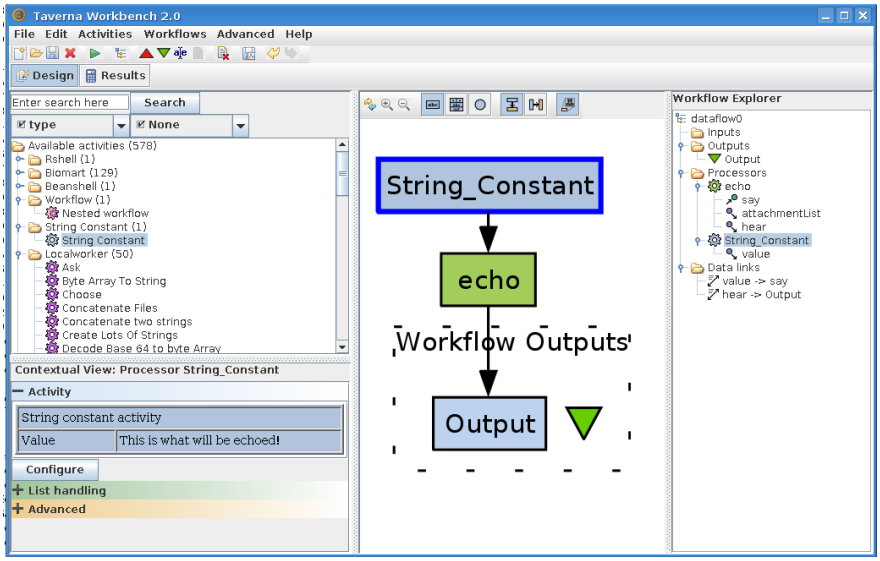


Figure 3.7: A completely functional (albeit small) workflow in Taverna that addresses the Echo Web Service prepared with HED.

check for the programmer's view on a WSDL file is to load it from within the workflow suite Taverna[†].

The *activities* menu in Taverna offers the item *New Activity*. It will request the specification of an URL or a local path that points to a WSDL file. The services presented in there will then be added to the list of WSDL resources. So will the *echo* workflow that was just prepared with Eclipse. When it is dragged to the canvas in the middle, as shown in figure 3.6, it appears as an interactive object. To the right, the parameters of the operation are displayed.

With a string constant as a constant input added, alternatively one could link a configurable workflow input as input to the echo Web Service, and the workflow's output linked as a sink to the output of the workflow, the workflow is complete. It can now be executed, presuming that the localhost is indeed running arched with the echo service as described in section **??**.

When comparing the GUI representation of Eclipse as shown with in figure 3.5 or that of the just shown Taverna representation with the raw WSDL of listing 3.1, it becomes apparent, that the raw form of the WSDL, as shown below, is more complicated. It seems to invent nested structures that are just not present, since it is only simple strings that are exchanged. The good news here is, that one would have reached exactly that more complicated looks when following the original setup without assignment of the simple

[†]Taverna is available from `http://taverna.sf.net`. ARC offers a plugin that allows the direct communication of Taverna with resources runnign ARC-0. An interface to ARC-1 is still pending. In principle, there is none needed. In practice, we still need more practice to decide.

type 'string'. WSDL needs special types for every operation. And eclipse provides them, even when the input types are shown in a more simple manner.

This has the advantage, that you can use this WSDL file to describe the regular ARC echo service, even though the original programmer might not have used Eclipse to create it. Or has he? Gabor?

3.3 What else to do with WSDL

There are several efforts to organise the world of web services and their interactions. Taverna was already mentioned above. Amongst the most advanced scientific community with respect to Web Services are the Bioinformaticians. Topics that come to mind are:

- BioMoby www.biomoby.org
- ^{m}yGrid and ^{m}yExperiment www.myexperiment.org
- Distributed Annotation System (DAS) http://www.biodas.org

But this is left for a future version of this tutorial... and/or possibly Hajo's Bachelor thesis.

Listing 3.1: WSDL file for Echo created with Eclipse.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://
    localhost:60000/echo/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://
    www.w3.org/2001/XMLSchema" name="echo" targetNamespace="http://localhost:60000/echo/">
  <wsdl:types>
    <xsd:schema targetNamespace="http://www.example.org/echo/">
      <xsd:element name="echo">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="in" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="echoResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="out" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="echoRequest">
    <wsdl:part name="say" type="xsd:string">
        <wsdl:documentation>This parameter specifies what should be said</wsdl:documentation
            ></wsdl:part>
  </wsdl:message>
  <wsdl:message name="echoResponse">
    <wsdl:part name="hear" type="xsd:string">
        <wsdl:documentation>This parameter shows the echo</wsdl:documentation></wsdl:part>
  </wsdl:message>
  <wsdl:portType name="echo">
    <wsdl:documentation>In this portType, the operations of the web service are brought
        together. There could be multiple.
For instance, the echo web service could buffer the input with one operation, and release it
     with another.
It would then work like a stack</wsdl:documentation>
    <wsdl:operation name="echo">
      <wsdl:documentation></wsdl:documentation>
      <wsdl:input message="tns:echoRequest">
        <wsdl:documentation>This is the input to this web service</wsdl:documentation></
            wsdl:input>
      <wsdl:output message="tns:echoResponse">
        <wsdl:documentation>This is the output of the web service</wsdl:documentation></
            wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="echoSOAP" type="tns:echo">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="echo">
      <soap:operation soapAction="http://www.example.org/echo/NewOperation"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="echo">
    <wsdl:documentation>An example for a WSDL file that describes the echo Web Service of
        ARC</wsdl:documentation>
    <wsdl:port binding="tns:echoSOAP" name="echoSOAP">
      <wsdl:documentation>Echo may be on various URLs with various ports. Here we have
          chosen port 60000. An often seen alternative is 50000</wsdl:documentation>
      <soap:address location="http://localhost:60000/echo"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

4 The Echo Web Service

The Echo Web Service was already presented in the introduction, basically as the ground truth to learn if the installation of the HED is functional. And it was frequently referenced in the previous chapter on WSDL. In this section, we will implement our own version of the Echo Web Service.

Its task is to return the content of the received message back to the client. As a special feature that we introduce solely for educatory purposes, the service provides two additional operations called “ordinary” and “reverse”. Depending on the operation chosen by the client, the message gets returned unmodified or reversed. The goal of this chapter is to deepen the knowledge on message processing.

The section starts with another roundup on the WSDL for the Echo service, continues with the implementation of the server and finally describes the client.

4.1 Description with WSDL

The WSDL file that specifies the here presented Echo Web Service is shown in Listing 4.1.

Listing 4.1: WSDL file describing the echo service. Filename: echo.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="urn:echo"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:echo="urn:echo">

<!-- Definition of the types                                    -->
<!-- Data types of the service                                  -->
<wsdl:types>
    <xsd:schema targetNamespace="urn:echo">
        <xsd:import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
        <!-- xsd definition of the request -->
        <xsd:simpleType name="say">
            <xsd:attribute name="operation" type="xsd:string" use="required"/>
            <xsd:restriction base="xsd:string"/>
        </xsd:simpleType>
        <xsd:complexType name="echoRequest">
            <xsd:sequence>
                <xsd:element name="say" type="echo:say" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>

        <!-- xsd definition of the response -->
        <xsd:simpleType name="hear">
            <xsd:restriction base="xsd:string"/>
        </xsd:simpleType>
        <xsd:complexType name="echoResponse">
            <xsd:sequence>
                <xsd:element name="hear" type="echo:hear" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
        <xsd:element name="echoRequest" type="echo:echoRequest"/>
        <xsd:element name="echoResponse" type="echo:echoResponse"/>

        <!-- Empty message for the fault -->
        <element name="echoFault" type="echoFault">
            <complexType>
                <sequence/>
            </complexType>
        </element>

    </xsd:schema>
</wsdl:types>

```

```
50 <!-- Definition of the messages                                    -->
51 <!-- Available messages of the service                            -->
52 <wsdl:message name="echoRequest">
53     <wsdl:part name="body" element="echo:echo"/>
54 </wsdl:message>
55 <wsdl:message name="echoResponse">
56     <wsdl:part name="body" element="echo:echoResponse"/>
57 </wsdl:message>
58
59 <!-- Definition of the ports                                      -->
60 <!-- The way to use the messages                                  -->
61 <wsdl:portType name="echo">
62     <wsdl:operation name="echo">
63         <wsdl:input name="echoRequest" message="echo:echoRequest"/>
64         <wsdl:output name="echoResponse" message="echo:echoResponse"/>
65         <wsdl:fault message="echo:echoFault"/>
66     </wsdl:operation>
67 </wsdl:portType>
68
69 <!-- Definition of the binding                                    -->
70 <!-- Specification of the message format and protocol to be used -->
71 <wsdl:binding name="echo" type="echo:echo">
72     <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
73     <wsdl:operation name="echo">
74         <soap:operation soapAction="echo"/>
75         <wsdl:input name="echoRequest">
76             <soap:body use="literal"/>
77         </wsdl:input>
78         <wsdl:output name="echoResponse">
79             <soap:body use="literal"/>
80         </wsdl:output>
81     </wsdl:operation>
82 </wsdl:binding>
83
84 <!-- Defines the endpoint of the service                          -->
85 <wsdl:service name="echo">
86     <wsdl:port name="echo" binding="echo:echo">
87         <soap:address location="http://localhost:60000/echo">
88     </wsdl:port>
89 </wsdl:service>
90
91 </definitions>
```

4.2 Implementation of the Server

This section presents the implementation of the service defined in the WSDL file of the previous section and shown in Listing 4.2. For the user of a web service, it is not important to know the programming language that this was implemented in. This tutorial uses the C++ libraries of HED. Though ARC also provides everything that one would need to implement the service with a scripting language like Python. A later version of this tutorial hopefully describes both APIs side by side.

4.2.1 Communication with SOAP: The SOAP Fault Message

A SOAP fault message is always located within the body element and can appear only once. It indicates error messages in case the port type *solicit-response* (input, output, fault) has been chosen. An element called *fault* encapsulates different types of elements that describe the fault. These elements are listed in Table 4.1. The fault code describes the generic class of the fault. The ARC implementation assigns each

Table 4.1: Elements of the fault message.

Element	Description
faultcode	Code which shall identify the fault.
faultstring	A human readable explanation of the fault.
faultactor	Information about who caused the fault to happen.
detail	Holds application specific error information.

fault a certain number. The fault code along with its description is described within the Table 4.2

Table 4.2: Possible fault codes.

Fault code	Error	Description
0	**undefined**	—
1	**unknown**	Reason for failure couldn't be identified.
2	**VersionMismatch**	Found an invalid namespace for the SOAP Envelope element.
3	**MustUnderstand**	The processing of the SOAP header is optional unless the flag mustUnderstand is set "true". If, according to that case, the SOAP service is not able to understand the header a fault message with the faultcode MustUnderstand is returned.
4	**Client**	The request message was incorrectly formed or contained incorrect information.
5	**Server**	There was a problem with the server so the message could not proceed
6	**DataEncodingUnknown**	The message couldn't be processed due to encoding problems

For more information additional tutorials can be found in the appendix or check the internet address http://www.w3.org/TR/wsdl.

4.2.2 The Service in C++

The header file will be set aside since it contains only redundant information. It can be reread in the source directory that is accompanying the tutorial. In comparsion with Time Web Service, basically three changes are required. The first change concerns the constructor and can be found at line 52. Two strings are extracted of the server configuration file which are later used to envelope the message before it is returned. Furthermore, a new method *makeFault* in line 66 has been introduced which is able to create a fault messages. Within the method *process* two new code fragments are now extracting and analyzing the incoming message, to be seen at line 100 and 110. In case something unexpected happens either will create a SOAP fault message.

Listing 4.2: Implementation of the Echo Web Service.

```
1  #include <arc/loader/Plugin.h>
2  #include <arc/message/PayloadSOAP.h>
3
4  #include <stdio.h>
5
6  #include "echoservice.h"
7
8
9  /**
10  * Initializes the echo service and returns it.
11  */
12 static Arc::Plugin* get_service(Arc::PluginArgument* arg)
13 {
14     Arc::ServicePluginArgument* mccarg = dynamic_cast<Arc::ServicePluginArgument*>(arg);
15     if(!mccarg) return NULL;
16     return new ArcService::EchoService((Arc::Config*)(*mccarg));
17 }
18
19 /**
20  * This PLUGINS_TABLE_NAME is defining basic entities of the implemented .
21  * service. It is used to get the correct entry point to the plugin.
22  */
23 Arc::PluginDescriptor PLUGINS_TABLE_NAME[] = {
24     {
25         "echo",                /* Unique name of plugin in scope of its kind   */
```

```
26          "HED:SERVICE",          /* Type/kind of plugin                          */
27          1,                      /* Version of plugin (0 if not applicable)      */
28          &get_service            /* Pointer to constructor function              */
29      },
30      { NULL, NULL, 0, NULL }     /* The array is terminated by element           */
31                                  /* with all components set to NULL              */
32  };
33
34
35  using namespace Arc;
36
37  namespace ArcService
38  {
39
40      /**
41      * Constructor. Calls the super constructor, initializing the logger and
42      * setting the namespace of the payload. Furthermore the prefix and suffix will
43      * extracted out of the Config object (Elements were declared inside the HED
44      * configuration file).
45      */
46      EchoService::EchoService(Arc::Config *cfg) : Service(cfg),logger(Logger::rootLogger, "
            Echo")
47      {
48          // Setting the namespace of the outgoing payload
49          ns_["echo"]="urn:echo";
50
51          // Extract prefix and suffix specified in the HED configuration file
52          prefix_=(std::string)((*cfg)["prefix"]);
53          suffix_=(std::string)((*cfg)["suffix"]);
54      }
55
56      /**
57      * Deconstructor. Nothing to be done here.
58      */
59      EchoService::~EchoService(void)
60      {
61      }
62
63      /**
64      * Method which creates a fault payload
65      */
66      Arc::MCC_Status EchoService::makeFault(Arc::Message& outmsg, const std::string &reason)
67      {
68          logger.msg(Arc::WARNING, "Creating fault! Reason: \"%s\"",reason);
69          // The boolean true indicates that inside of PayloadSOAP,
70          // an object SOAPFault will be created inside.
71          Arc::PayloadSOAP* outpayload = new Arc::PayloadSOAP(ns_,true);
72          Arc::SOAPFault* fault = outpayload->Fault();
73          if(fault) {
74              fault->Code(Arc::SOAPFault::Sender);
75              fault->Reason(reason);
76          };
77          outmsg.Payload(outpayload);
78
79          return Arc::MCC_Status(Arc::STATUS_OK);
80      }
81
82      /**
83      * Processes the incoming message and generates an outgoing message.
84      * Returns the incoming string ordinary or reverse - depending on the operation
85      * requested.
86      * @param inmsg incoming message
87      * @param inmsg outgoing message
88      * @return Status of the result achieved
89      */
90      Arc::MCC_Status EchoService::process(Arc::Message& inmsg, Arc::Message& outmsg)
91      {
92          logger.msg(Arc::DEBUG, "Echoservice started...");
93
94          /** Both input and output are supposed to be SOAP */
95          Arc::PayloadSOAP* inpayload  = NULL;
96          Arc::PayloadSOAP* outpayload = NULL;
97          /** */
98
99          /**  Extracting incoming payload */
100         try {
101             inpayload = dynamic_cast<Arc::PayloadSOAP*>(inmsg.Payload());
102         } catch(std::exception& e) { };
103
104         if(!inpayload) {
105             logger.msg(Arc::ERROR, "Input is not SOAP");
106             return makeFault(outmsg, "Received message was not a valid SOAP message.");
107         };
108         /** */
109
110         /** Analyzing and execute request */
111         Arc::XMLNode requestNode  = (*inpayload)["echo:echoRequest"];
```

```
112         Arc::XMLNode sayNode      = requestNode["echo:say"];
113         std::string operation = (std::string) sayNode.Attribute("operation");
114         std::string say       = (std::string) sayNode;
115         std::string hear      = "";
116         logger.msg(Arc::DEBUG, "Say: \"%s\"  Operation: \"%s\"",say,operation);
117
118         //Compare strings with constants defined in the header file
119         if(operation.compare(ECHO_TYPE_ORDINARY) == 0)
120         {
121             hear = prefix_+say+suffix_;
122         }
123         else if(operation.compare(ECHO_TYPE_REVERSE) == 0)
124         {
125             int len = say.length ();
126             int n;
127             std::string reverse = say;
128             for (n = 0;n < len; n++)
129             {
130                 reverse[len - n - 1] = say[n];
131             }
132             hear = prefix_+ reverse +suffix_;
133         }
134         else
135         {
136             return makeFault(outmsg, "Unknown operation. Please use \"ordinary\" or \"
                    reverse\"");
137         }
138         /** */
139
140         /** Create response */
141         outpayload = new Arc::PayloadSOAP(ns_);
142         outpayload->NewChild("echo:echoResponse").NewChild("echo:hear")=hear;
143         outmsg.Payload(outpayload);
144
145         /** */
146
147         logger.msg(Arc::DEBUG, "Echoservice done...");
148         return Arc::MCC_Status(Arc::STATUS_OK);
149     }
150
151 }//namespace
```

Again, the HED is configured by an XML file and launched in the same manner as introduced before. The server configuration file is shown in Listing 4.3. The only change concerns the lines 37 and 38 which specify the *prefix* and *suffix*. The parameters will be extracted in the constructor of the service and explain nicely how settings can be passed from the server configuration file towards the service.

4.3 Implementation of the Client

With the interface of the server gaining complexity, so is the code for the client - for two reasons. Firstly, because the additional arguments need to be filled. Secondly, because of the extra effort to interact with the (here human) user to collect the data which needs to be forwarded to the Echo Web Service. The source code of the client is shown Listing 4.4. The parameters are now passed by the command line, to be seen in the lines following line 15. That code is independent from the ARC library. The creation of the SOAP client is unchanged from the Echo client's implementation and the request to the server is created in the lines subsequent to line 62. It is processed in the code after line 69 in which also the response is received. The checks of the response have expanded by an additional condition which tests for a fault message, see line 75.

4.3.1 The Client in C++

Listing 4.4: Implementation of the client program.

```
1 #include <arc/ArcLocation.h>
2 #include <arc/message/MCC.h>
3 #include <arc/client/ClientInterface.h>
4 #include <arc/client/UserConfig.h>
5
6 #include <iostream>
7 #include <string>
8
9 using namespace Arc;
```

Listing 4.3: Server configuration file of the Echo Web Service

```
<?xml version="1.0"?>
<ArcConfig
  xmlns="http://www.nordugrid.org/schemas/ArcConfig/2007"
  xmlns:tcp="http://www.nordugrid.org/schemas/ArcMCCTCP/2007"
  xmlns:tls="http://www.nordugrid.org/schemas/ArcMCCTLS/2007"
  xmlns:echo="urn:echo_config"
>
    <Server>
    <Pidfile>/var/run/arched.pid</Pidfile>
    <Logger level="VERBOSE">/var/log/arched.log</Logger>
    </Server>
    <ModuleManager>
        <Path>/tmp/arc/tutorial/lib/</Path>
        <Path>/usr/lib/arc/</Path>
    </ModuleManager>
    <Plugins><Name>mcctcp</Name></Plugins>
    <Plugins><Name>mcctls</Name></Plugins>
    <Plugins><Name>mcchttp</Name></Plugins>
    <Plugins><Name>mccsoap</Name></Plugins>
    <Plugins><Name>echoservice</Name></Plugins>
    <Chain>
        <Component name="tcp.service" id="tcp">
            <next id="http"/>
            <tcp:Listen><tcp:Port>60000</tcp:Port></tcp:Listen>
        </Component>
        <Component name="http.service" id="http">
            <next id="soap">POST</next>
        </Component>
        <Component name="soap.service" id="soap">
            <next id="plexer"/>
        </Component>
        <Plexer name="plexer.service" id="plexer">
            <next id="echo">^/echo$</next>
        </Plexer>
        <Service name="echo" id="echo">
            <next id="echo"/>
            <echo:prefix>[ </echo:prefix>
            <echo:suffix> ]</echo:suffix>
        </Service>
    </Chain>
</ArcConfig>
```

```

int main(int argc, char** argv)
{

    /** Get arguments passed by the command line*/
    if(argc != 4){
        printf("Usage:\n");
        printf("  echoclient url type message\n");
        printf("\n");
        printf("The first argument defines the URL of the service.\n");
        printf("The second argument the type of echo which shall be performed (ordinary or
            reverse).\n");
        printf("The third agrument holds the message to be transmitted.\n");
        printf("\n");
        printf("Example:\n");
        printf("  ./echoclient http://localhost:60000/echo ordinary text_to_be_transmitted\n
            ");
        return -1;
    }

    std::string urlstring(argv[1]);
    std::string type(argv[2]);
    std::string message(argv[3]);
    /** */

    // Initiate the Logger and set it to the standard error stream
    Arc::Logger::rootLogger.setThreshold(Arc::WARNING);
    Arc::Logger logger(Arc::Logger::getRootLogger(), "echo");
    Arc::LogStream logcerr(std::cerr);
    Arc::Logger::getRootLogger().addDestination(logcerr);
    Arc::Logger::rootLogger.setThreshold(Arc::WARNING);

    // Set the ARC installation directory
    Arc::ArcLocation::Init("/usr/lib/arc");
    setlocale(LC_ALL, "");

    // Load user configuration (default ~/.arc/client.xml)
    std::string conffile;
```

```
46      Arc::UserConfig usercfg(conffile);
47      if (!usercfg) {
48          printf("Failed configuration initialization\n");
49          return 1;
50      }
51
52      // Basic MCC configuration
53      Arc::MCCConfig cfg;
54      Arc::URL service(urlstring);
55
56      // Creates a typical SOAP client based on the
57      // MCCConfig and the service URL
58      Arc::ClientSOAP client(cfg, service);
59
60
61      // Defining the namespace and create a request
62      Arc::NS ns("echo", "urn:echo");
63      Arc::PayloadSOAP request(ns);
64      XMLNode sayNode = request.NewChild("echo:echoRequest").NewChild("echo:say") = message;
65      sayNode.NewAttribute("operation") = type;
66
67
68      // Process request and get response
69      Arc::PayloadSOAP *response = NULL;
70      Arc::MCC_Status status = client.process(&request, &response);
71
72      if (!response) {
73          printf("No SOAP response");
74          return 1;
75      }else if(response->IsFault()){
76          printf("A SOAP fault occured:\n");
77          std::string hoff = (std::string) response->Fault()->Reason();
78          printf("  Fault code:   %d\n", (response->Fault()->Code()));
79          printf("  Fault string: \"%s\"\n", (response->Fault()->Reason()).c_str());
80          /** 0  undefined,             4  Sender,   // Client in SOAP 1.0
81              1  unknown,               5  Receiver, // Server in SOAP 1.0
82              2  VersionMismatch,       6  DataEncodingUnknown
83              3  MustUnderstand,                                    **/
84
85          std::string xmlFault;response->GetXML(xmlFault, true);  printf("\%s\n\n", xmlFault.
                c_str());
86          return 1;
87      }
88
89      // Test for possible errors
90      if (!status) {
91          printf("Error %s",((std::string)status).c_str());
92          if (response)
93              delete response;
94          return 1;
95      }
96
97
98      std::string answer = (std::string)((*response)["echo:echoResponse"]["echo:hear"]);
99      std::cout << answer <<std::endl;
100
101     delete response;
102
103     return 0;
104 }
```

The usage of the client is presented in Listing 4.5. Three cases of Echo Web Service responses are displayed: ordinary operation, reverse operation and SOAP fault due to an unknown operation.

Listing 4.5: Three diffrent invokations of the client program.

```
$ ./echoclient http://localhost:60000/echo ordinary text_to_be_transmitted
[ text_to_be_transmitted ]
$ ./echoclient http://localhost:60000/echo reverse text_to_be_transmitted
[ dettimsnart_eb_ot_txet ]
$ ./echoclient http://localhost:60000/echo re2verse text_to_be_transmitted
A SOAP fault occured:
  Fault code:   4
  Fault string: "Unknown operation. Please use "ordinary" or "reverse""
```

A few examples of exchanged messages are shown in the Listings 4.6, 4.7 and 4.8. Listing 4.6 illustrates a request message using the operation *reverse* while Listing 4.7 illustrates the corresponding response message. The last Listing 4.8 shows the fault message which is created by the service after the client requested an unknown operation.

Listing 4.6: Request message created by the client.

```
1 <soap-env:Envelope xmlns:echo="urn:echo" xmlns:soap-enc="http://schemas.xmlsoap.org/soap/
      encoding/" xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http:
      //www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
2   <soap-env:Body>
3     <echo:echoRequest>
4       <echo:say operation="reverse">text_to_be_transmitted</echo:say>
5     </echo:echoRequest>
6   </soap-env:Body>
7 </soap-env:Envelope>
```

Listing 4.7: Response message created by the service.

```
1 <soap-env:Envelope xmlns:echo="urn:echo" xmlns:soap-enc="http://schemas.xmlsoap.org/soap/
      encoding/" xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http:
      //www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
2   <soap-env:Body>
3     <echo:echoResponse>
4       <echo:hear>[ dettimsnart_eb_ot_txet ]</echo:hear>
5     </echo:echoResponse>
6   </soap-env:Body>
7 </soap-env:Envelope>
```

Listing 4.8: Fault message created by the service.

```
1 <soap-env:Envelope xmlns:echo="urn:echo" xmlns:soap-enc="http://schemas.xmlsoap.org/soap/
      encoding/" xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http:
      //www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
2   <soap-env:Body>
3     <soap-env:Fault>
4       <soap-env:faultcode>soap-env:Client</soap-env:faultcode>
5       <soap-env:faultstring>Unknown operation. Please use "ordinary" or "reverse"</soap-
            env:faultstring>
6     </soap-env:Fault>
7   </soap-env:Body>
8 </soap-env:Envelope>
```

5 The TLS Echo Web Service

The Web Services of the previous chapters have one major drawback: the communication between the two endpoints is insecure. For that reason the current and the next chapters will discuss how to create a secure Web Service. In general, security in a network can be subdivided into the following categories: [**?**]

- **Confidentiality** — Protection of the data against passive attacks (such as publicise message content)
- **Authentication** — Verification of the identity of the communication partners.
- **Integrity** — Protection against interception and manipulation, replay, insertion, etc. of a message.
- **Non-repudiation** — Preventing the sender to repudiate the message transmitted by him.
- **Access control** — Restrict the access to ressources.
- **Availability** — Ensure a system is always usable which is challenged by various attacks.

Ideally, all six categories should have been achieved by such an application (though the last category, availability, is difficult to realise). In the present chapter, the security will be put into practice by extending the protocol stack with an additional security layer called TLS (Transport Layer Security). Due to the location between the TCP and the HTTP layer, the layer is only capable to limit the access of the grid to a certain set of users. Once the user has passed the layer sucessfully, all resources provided by the HED are available. This kind of behaviouor can be seen as to be very coarse grained. Nevertheless, it provides the security categories: confidentiality, authentication, integrity, non-repudiation and access control.

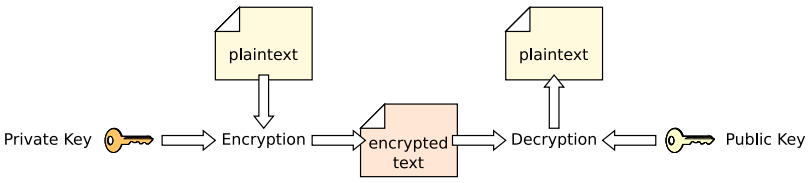
The next section gives a basic understanding of the working principle of the TLS. In order to enable newcomers an entry to this area, emphasis is put on the security concept and not on an exact description of the technical realisation.

5.1 Transport Layer Security

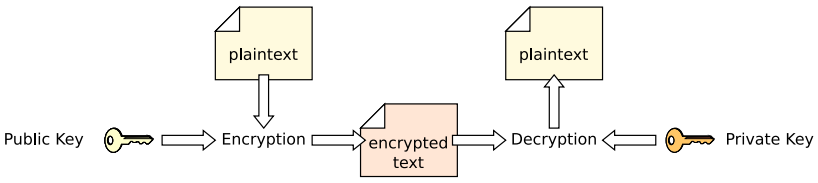
The TLS is a protocol which provides a secure connection between two endpoints. It is based directly on the TCP/IP layers and uses typically an asymmetric encryption to establish the connection (alternatively a symmetric pre-shared key may be used). After the communication has been successfully established, both participants are switching to another encryption method which is based on a new negotiated symmetric key.

Symmetric and asymmetric encryption are the two main classes of cryptographic algorithms. In case of symmetric encryption, both participants are holding the same secret key to encrypt and decrypt a plain text. The principle is to reverse the process of encryption in order to decrypt the text. Symmetric encryption can be ranked as being very secure but its downside is that the key distribution is very difficult in practice. The key has to be shared previously, ideally over a secure channel which may either be a complete different medium (i.e. a letter or speech) or a channel which has been encrypted with another secret key. Asymmetric encryption follows a different functional principle. While the symmetric encryption utilise one key, the asymmetric ecryption is based on two keys: a private key and public key. As the names imply, the private key is kept secret and is never revealed to others while the public key is available for everyone. Messages encrypted with the public key can only be decrypted with the private key and conversely, messages encrypted with the private key can only be decrypted with the public key. The Figure 5.1 illustrates the circumstances of the case.

At startup of a communication process, A (Alice) who desires a secure connection transmits her public key to B (Bob). Bob is now able to encrypt his messages such that only Alice is able to decrypt them. As one can see, this first approach already provides confidentiality and integrity of messages. But neither non-repudiation, authentication and consequently nor access control are established because the identity



(a) A plain text which has been encrypted with the private key can only be decrypted with the public key. If the receivers is sure that the public key is related to a certain identity, the private key can be used to sign the messages. Only the owner of the private key is able to create encrypted messages which can be decrypted with the public key.



(b) A plain text which has been encrypted with the public key can only be decrypted with the private key. Using asymmetric keys in that manner offers the possiblity to send encrypted messages to the owner of the private key.

Figure 5.1: Usage of asymmetric encryption.

of Alice is not proven. In order to be able to check identities, so called certificates have been invented. Certificates render the possibility to validate the identity of its owner. They are signed by a certificate authority (CA), the identity of which can be verified by another CA or that is pre-configured to be trusted.

A certificate is composed of the identity of its owner, its public key, the name of the CA who signed the certificate and the signature of the CA which is a hash-value of the certificate encrypted with the private key of the CA. Furthermore, the algorithms used to create the certificate are added. If a certificate can be resolved to a trusted CA, the identity of the certificate owner is considered to be proven. In general, a set of trusted CAs is already defined in the operation system. To obtain a certificate, a certificate request has to be submitted to a CA. The CA verifies the identity of the requesting person and the requested certificate will be signed. Figure 5.2 shows the procedure of gaining a certificate. For instructions on the actual commands on the UNIX shell, you may follow instructions on http://ca.nordugrid.org. In the

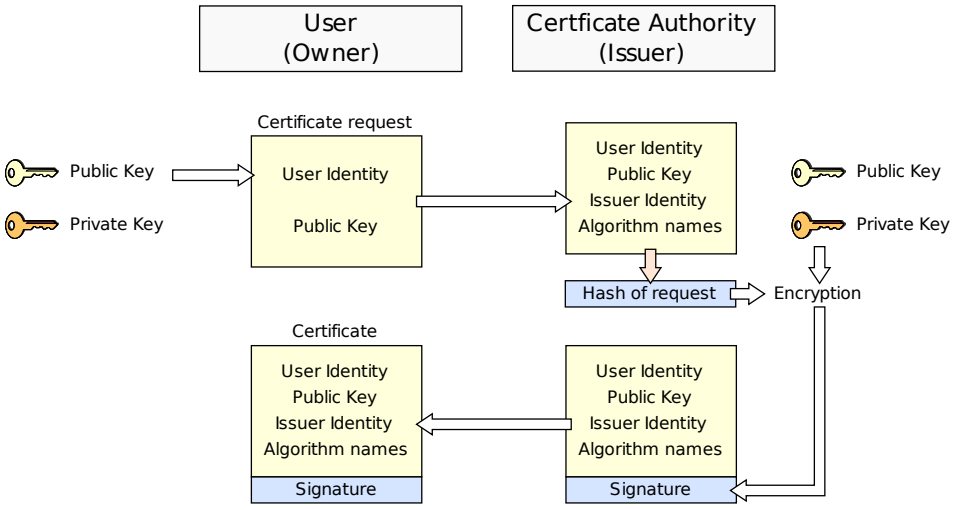


Figure 5.2: Procedure of certificate signing.

first step, a private and a public key will be generated by the requester. The public key together with a textual description of the identity depict the request which will be transmitted to a CA. After the CA successfully verified the validity of the request, the identity of the CA and the name of the used encryption algorithms are added. The collected data will represent the plain text of the certificate. In order to ensure the integrity of the text, a hash value will be created. This is a fixed-sized short string which is generated by an algorithm based on an arbitrary long given text. The algorithm is designed such that a small change in the text will cause the hash value to change almost like a random function. Thus, the hash value can be used a fingerprint of the text. The CA signs the plain certificate by encrypting its hash value with its private key. Finally the signed certifcate will be returned to the requester. If an outside person wants to verify the certificate owner, the encrypted hash value has to be decrypted by the public key of the CA and a hash value has to be created with the same algorithm in order to compare them.

Figure 5.3 shows a usecase in which Alice wants to resolve the identity of Bob. Alice submits her request to Bob along with a challange consisting of a random number n. Bob encrypts the number with his private key and returns it together with his certificate. The random number ensures that it is not possible to repeat a message (integrity) and to prove that the communication partner holds the private key. Alice uses the public key of Bob to decrypt the challenge. If the decrypted number is equal to the transmitted number, Alice can reason that the identity of Bob is to be trusted in case she is trusting the CA which has signed Bobs certificate.

In the given example, Alice only trusts the CA 2 but not CA 1. To resolve the identity of CA 1, Alice establishes a connection to CA 1 and submits another random number n as a challenge. The CA 1 encrypts the challenge with its private key and sends the result along with its certificate back to Alice. Alice successfully compares the returned number. Additionally she validates the certificate of CA 1 with the pre-configured certificate of CA 2.. Since the certificate was signed by CA 2 which is pre-configured to be trusted, she also trusts CA 1 and in conclusion the identity of B. In summary, Alice knows she is talking to Bob. If Bob also wants to be sure who he's talking to, then the same procedure has to be done to confirm the identity of Alice. The TLS mode in which both endpoints are ensuring themselves to whom they are talking to is called *bilateral connection mode*.

When the client is aware of the server's authentity, the confidential treatment of its data is guaranteed such that one can be sure the data won't be revealed to a third party (server-sided confidentiality). On the other side, if the server is aware of the client authentity, the access to services and resources can be controlled (access control). Thus, in case of services, the results will only be passed back to the right person (client-sided confidentiality). As one can see confidentiality, authentication, integrity and non-repudiation are now established.

Certificates have several advantages. They are scaling very good with the amount of users because CAs can create new sub-CAs such that the load is balanced (certificates of a CA may be cached). Furthermore, no critical information is exchanged for the establishment of a connection (in particular: no secret key is shared which may get intercepted). Instead, a set of pre-configured CAs which will be in general provided by the operation system is needed. In respect of ARC it is obvious that the access to the grid shall be under the control of its administrator. To do so, the administrators have to create their own CAs which are in charge of creating certificates for new users. The identity of the created CA has to be added to the directory of pre-configured CAs. Many universities have prepared local certificate authorities. And every user can establish his own. Certificates in general, however, have found entry to regular computer users, for instance with the FernUniversity Hagen (https://ca.fernuni-hagen.de/) who use that technology for their remote students to hand in exercises.

A CA which controls access to a grid can be considered as a virtual organsiation (VO). It may involve organisations close to the leading organisation as well as structure which are not formally associated with it. In most cases the basic idea is to share ressources.

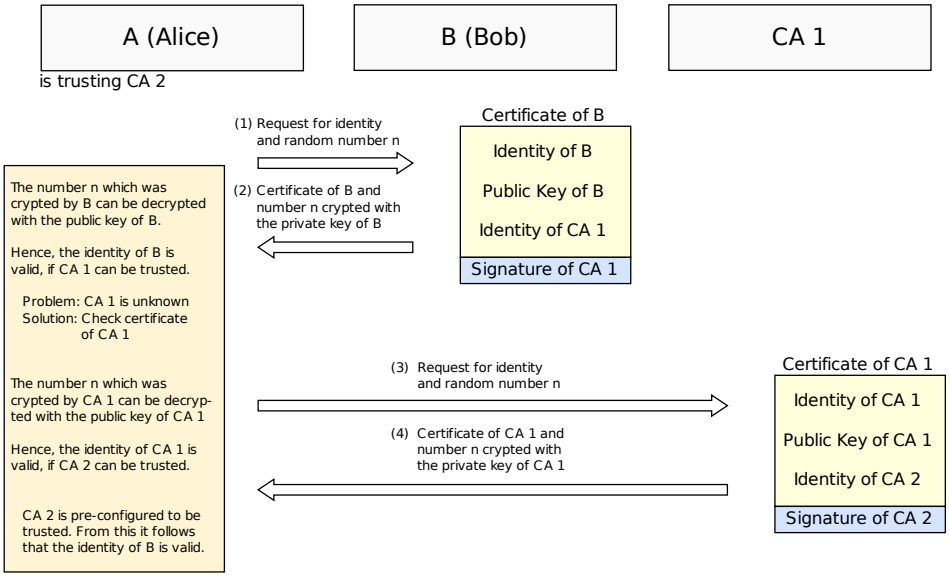


Figure 5.3: Usecase in which Alice resolves the identity of Bob. In order to resolve the identity of Bob, Alice sends a request for Bob's identity along with a newly created random number to Bob. Bob encrypts the received number using his private key and returns his public key to Alice. Due to the random number can only be encrypted by the owner of the private key, Alice is able ot verify that the certificate was transmitted by his owner. Since the certificate was signed by the CA 1, which she is not per-configured to be trust, Alice needs to resolve also the identity of CA 1. The request is done in the same manner, but this time the certificate is signed by the trusted CA 2. Due to the fact she is now able to trust CA 1, Alice can now also trust the identity of Bob.

5.2 Service

In order to create a secure connection we hence need two certificates: a client certificate and a service certificate. The CA of the certificate has to be pre-configured on the counterpart and furthermore, the owner of the certificate has to hold the fitting private key. For the given example X.509-certificates in the PEM format will be used which are standardised by the ITU-T. Altogether six files are needed:

clientCA — The CA which guarantees for the identity of the client
clientCERT — The certificate of the client.
clientKEY — The private key of the client (should be kept in safe custody).

serverCA — The CA which guarantees for the identity of the server
serverCERT — The certificate of the server.
serverKEY — The private key of the server (should be kept in safe custody).

To get this service running, the six files have to be distributed. The server should hold the files: serverCERT.pem, serverKEY.pem and clientCA.pem. The client on the other side should own the files: clientCERT.pem, clientKEY.pem and serverCA.pem. Within the source code directory that accompanies this document certificates with corresponding names are prepared. Due to certificates have a limited period of validity, these certificates may no longer be accepted by the communication parter. One has to create new certificates to ensure the proper work of the TLS Web Service.

Several tools are available which allow the handling for more advanced UNIX users. In order to create a first basic set of certificates signed by a self created CA, the shell scripts *createMasterCA.sh* and *createSlaveCert.sh* can be found within the directory *src/services/tlsechoservice/certFactory* may be used. The script *createMasterCA.sh* will create a new CA and will overwrite older ones. The script *createSlaveCert.sh* creates several certificates signed by the CA.

The TLS Echo Web Service implementation is almost the same as the one presented in the previous chapter. Merely the Class name and the namespace of the service have been changed. The only modification is done within the server configuration file which is displayed in Listing 5.1. The first change is in line 19. As to be seen, another plugin called *mcctls* will be loaded by the *ModuleManager*. The second change is the new element which has been placed between the TCP and the HTTP component, see line 28. Within the TLS element the certificates can be declared using the elements *KeyPath*, *CertificatePath*, *CACertificatePath* and *CACertificatesDir*.

Listing 5.1: HED configuration file for the Arc intern echo service. Filename: arcecho_no_ssl.xml

```
<?xml version="1.0"?>
<ArcConfig
  xmlns="http://www.nordugrid.org/schemas/ArcConfig/2007"
  xmlns:tcp="http://www.nordugrid.org/schemas/ArcMCCTCP/2007"
  xmlns:tls="http://www.nordugrid.org/schemas/ArcMCCTLS/2007"
  xmlns:ut="http://www.nordugrid.org/schemas/UsernameTokenSH"
  xmlns:arcpdp="http://www.nordugrid.org/schemas/ArcPDP"
  xmlns:tlsecho="urn:tlsecho_config"
>
    <Server>
    <Pidfile>/var/run/arched.pid</Pidfile>
    <Logger level="VERBOSE">/var/log/arched.log</Logger>
    </Server>
    <ModuleManager>
        <Path>/tmp/arc/tutorial/lib/</Path>
        <Path>/usr/lib/arc/</Path>
    </ModuleManager>
    <Plugins><Name>mcctcp</Name></Plugins>
    <Plugins><Name>mcctls</Name></Plugins>
    <Plugins><Name>mcchttp</Name></Plugins>
    <Plugins><Name>mccsoap</Name></Plugins>
    <Plugins><Name>tlsechoservice</Name></Plugins>
    <Chain>
        <Component name="tcp.service" id="tcp">
            <next id="tls"/>
            <tcp:Listen><tcp:Port>60000</tcp:Port></tcp:Listen>
        </Component>
        <Component name="tls.service" id="tls">
            <next id="http"/>
            <KeyPath>./serviceKey.pem</KeyPath>
            <CertificatePath>./serviceCert.pem</CertificatePath>
            <CACertificatePath>./clientCA.pem</CACertificatePath>
        </Component>
        <Component name="http.service" id="http">
            <next id="soap">POST</next>
        </Component>
        <Component name="soap.service" id="soap">
            <next id="plexer"/>
        </Component>
        <Plexer name="plexer.service" id="plexer">
            <next id="tlsecho">^/tlsecho$</next>
        </Plexer>
        <Service name="tlsecho" id="tlsecho">
            <next id="tlsecho"/>
            <tlsecho:prefix>[ </tlsecho:prefix>
            <tlsecho:suffix> ]</tlsecho:suffix>
        </Service>
    </Chain>
</ArcConfig>
```

5.3 Client

In this section a new way to implement the client will be introduced. As already mentioned in section 2.2, the protocol stack of the client is built up in almost the same way as the one of the service. Indeed, the class *ClientSOAP* composes the stack with the same MCCs used for the service. Once this is known, it is self-evident to create the client in a simlar manner like the service with an own configuration file. Listing 5.2 shows the source code of the client program. The main difference to the simple Echo Client presented in the previous chapter is that now the information about the client protocol stack is encapsulated within a configuration file.

Listing 5.2: Client programm which is capable to load a configuration file

```
1  #include <arc/ArcConfig.h>
2  #include <arc/ArcLocation.h>
3  #include <arc/client/ClientInterface.h>
4  #include <arc/client/UserConfig.h>
5  #include <arc/message/MCCLoader.h>
6  #include <arc/message/MCC.h>
7
8  #include <iostream>
9  #include <string>
10 #include <fstream>
11
12 using namespace Arc;
13
14 int main(int argc, char** argv) {
15
16     /** Get arguments passed by the command line*/
17     if(argc != 4){
18         printf("Usage:\n");
19         printf("  tlsechoclient configuration type message\n");
20         printf("\n");
21         printf("The first argument specifies the HED configuration file to be loaded\n");
22         printf("The second argument the type of echo which shall be performed (ordinary or
               reverse).\n");
23         printf("The third agrument holds the message to be transmitted.\n");
24         printf("\n");
25         printf("Example:\n");
26         printf("  ./tlsechoclient clientHED.xml ordinary text_to_be_transmitted\n");
27         return -1;
28     }
29
30     std::string type(argv[2]);
31     std::string message(argv[3]);
32     /** */
33
34     /** Load the client configuration file into a string */
35     std::string xmlstring;
36     std::ifstream file(argv[1]);
37     if (file.good())
38     {
39         printf("Reading XML schema from %s\n",argv[1]);
40         file.seekg(0,std::ios::end);
41         std::ifstream::pos_type size = file.tellg();
42         file.seekg(0,std::ios::beg);
43         std::ifstream::char_type *buf = new std::ifstream::char_type[size];
44         file.read(buf,size);
45         xmlstring = buf;
46         delete []buf;
47         file.close();
48     }
49     else
50     {
51         std::cout << "File " << argv[1] << " not found." << std::endl;
52         return -1;
53     }
54     /** */
55
56     // Initiate the Logger and set it to the standard error stream
57     Arc::Logger logger(Arc::Logger::getRootLogger(), "arcecho");
58     Arc::LogStream logcerr(std::cerr);
59     Arc::Logger::getRootLogger().addDestination(logcerr);
60     Arc::Logger::rootLogger.setThreshold(Arc::WARNING);
61
62     // Set the ARC installation directory
63     std::string arclib("/usr/lib/arc");
64     Arc::ArcLocation::Init(arclib);
65
66     /** Load the client configuration */
67     Arc::XMLNode clientXml(xmlstring);
68     Arc::Config clientConfig(clientXml);
```

```
69      if(!clientConfig) {
70        logger.msg(Arc::ERROR, "Failed to load client configuration");
71        return -1;
72      };
73
74      Arc::MCCLoader loader(clientConfig);
75      logger.msg(Arc::INFO, "Client side MCCs are loaded");
76      Arc::MCC* clientEntry = loader["soap"];
77      if(!clientEntry) {
78        logger.msg(Arc::ERROR, "Client chain does not have entry point");
79        return -1;
80      };
81      /** */
82
83
84      /** Create and process message */
85      Arc::NS ns("tlsecho", "urn:tlsecho");
86      Arc::PayloadSOAP  request(ns);
87      Arc::PayloadSOAP* response = NULL;
88
89      // Due to the fact that the class ClientSOAP isn't used anymore,
90      // one has to prepare the messages which envelope the payload themselves.
91      Arc::Message reqmsg;
92      Arc::Message repmsg;
93      Arc::MessageAttributes attributes_req;
94      Arc::MessageAttributes attributes_rep;
95      Arc::MessageContext context;
96      repmsg.Attributes(&attributes_rep);
97      reqmsg.Attributes(&attributes_req);
98      reqmsg.Context(&context);
99      repmsg.Context(&context);
100
101     XMLNode sayNode = request.NewChild("tlsecho:tlsechoRequest").NewChild("tlsecho:say") =
            message;
102     sayNode.NewAttribute("operation") = type;
103     reqmsg.Payload(&request);
104
105     Arc::MCC_Status status = clientEntry->process(reqmsg,repmsg);
106     response = dynamic_cast<Arc::PayloadSOAP*>(repmsg.Payload());
107     /** */
108
109     if (!response) {
110         printf("No SOAP response");
111         return 1;
112     }else if(response->IsFault()){
113         printf("A SOAP fault occured:\n");
114         std::string hoff = (std::string) response->Fault()->Reason();
115         printf("  Fault code:   %d\n", (response->Fault()->Code()));
116         printf("  Fault string: \"%s\"\n", (response->Fault()->Reason()).c_str());
117         /** 0  undefined,             4  Sender,    // Client in SOAP 1.0
118             1  unknown,               5  Receiver, // Server in SOAP 1.0
119             2  VersionMismatch,       6  DataEncodingUnknown
120             3  MustUnderstand,                                          **/
121
122         std::string xmlFault;response->GetXML(xmlFault, true);  printf("\%s\n\n", xmlFault.
                c_str());
123         return 1;
124     }
125
126     // Test for possible errors
127     if (!status) {
128         printf("Error %s",((std::string)status).c_str());
129         if (response)
130             delete response;
131         return 1;
132     }
133
134
135     std::string answer = (std::string)((*response)["tlsecho:tlsechoResponse"]["tlsecho:hear"
            ]);
136     std::cout << answer <<std::endl;
137
138     delete response;
139 }
```

The file containing the client configuration is specified within the command line. Its content will be loaded in the code fragment starting at line 34. Later, in line 66, the configuration will be parsed into XML and than transfered into a chain of components which is wrapped by the object *loader*. In order to process a message, an entry point within the chain has to be disclosed. To implement a SOAP client, we are looking for the SOAP component. To get the proper component an operator of MCCLoader is used in line 76. The previously used class *ClientSOAP* has eased some workload which now has to be done within the client program. The payloads now have to be wrapped by message objects which have to be initialised with an

MessageAttribute object and a *MessageContext* object. After the payload has been wrapped within the message it gets transmitted in almost the same way like before, see lines following by line 84. The remainder of the source code is identical to the Echo Client.

6 Secure Echo Web Service

The TLS Echo Web Service, which was presented in the previous chapter, was able to control the access to the HED on a very low layer. For the administration of a large VO the kind of security may be too general. Thus, it is recommended to have a more fine-grained security. This chapter discusses how to install a more complex security such that it is possible to limit the access of certain resources to particular users and to constrain their rights on the workstation.

The security framework of ARC is based on the same concept of modularity as HED. To extend the security of a MCC or a service one only needs to modify the server configuration file. That is done by including a *SecHandler* element into the component or service which contains a PDP element. A small example of a chain component containing a *SecHandler* is shown in Listing 6.1. Within the element *Component*, which realises a HTTP layer, an element *SecHandler* is located. It contains three attributes. The attribute *name* is *arc.authz*, the *event* attribute is *incoming* and the *id* attribute which is *auth*. While *id* is only an optional parameter, the attribute *name* causes a certain *SecHandler* to be loaded. In case the attribute *name* is assigned to *arc.authz*, the SecHandler provides the functionality of authentication and is able to permit or to deny the access to the next layer. Each MCC or Service usually implements two queues of *SecHandlers* — one for incoming messages and one for outgoing messages. The attribute *event* attaches the present one to the incoming queue. *SecHandlers* are able to intervene the message flow by: manipulating the message content, manipulating the message attributes or to disrupt messages. Within one component, *SecHandlers* are executed sequentially in the order in which they were attached to the queue. If any of them fails, message processing fails as well [**?**].

In order to decide how the *SecHandler* shall modify the message, so called PDP (Policy Decision Point) are used. Within the example a PDP named *simplelist.pdp* is inserted within the *SecHandler* element. The mechanism of this PDP uses a file which contains a list of identities. If the current identity is on the list, the PDP returns a positive result to the *SecHandler*. In general more than one PDP can be defined within one *SecHandler*. The default behaviour in that case is to execute all PDPs sequentially until one fails such that a negativ result will be returned or all return a negativ result such that also a positive result will be passed to the *SecHandler*.

Listing 6.1: Example of a SecHandler within the component *http.service*

```
1 <Component name="http.service" id="http">
2     <next id="soap">POST</next>
3     <SecHandler name="arc.authz" event="incoming" id="auth">
4         <PDP name="simplelist.pdp" location="./clientlist.txt"/>
5     </SecHandler>
6 </Component>
```

Due to the fact, that presenting all *SecHandlers* and PDPs would go beyond the scope of an introducing tutorial, only a small set of them will be introduced here in detail. Nevertheless, the Tables 6.1 and 6.2 shall give a first overview about existing *SecHandlers* and *PDPs*. For further reading it is recommended to consult the document “Security framework of ARC1” written by Weizhong Qiang and Alexandr Konstantinov. The present chapter is closly related to it.

The Secure Echo Web Service which will be presented in this chapter, is using the *SecHandlers*: *identity.map* and *arc.authz*, and the *PDPs*: *allow.pdp*, *simplelist.pdp* and *arc.pdp*.

Table 6.1: List of built-in SecHandlers

identity.map	Maps the identity of the user to an identity of the workstation.
arc.authz	Grants access to the next layer.
delegation.collector	Processes a request for another service located on a different HED. A certificate, which has a delegation policy embedded, is transfered to the service running that *SecHandler*. Once the has been extracted and verified it will be transfered to the requested service. The policy is now available within the HED of the service such that the client is enabled to access the service.
usernametoken.handler	Generates the WS-Security username-token and adds it into the SOAP header of outgoing messages. In case of incoming messages the SecHandler extracts the WS-Security username-token out of it. The WS-Security standard extends SOAP and describes i.e. how to attach signatures and encryption header to SOAP messages.

Table 6.2: List of built-in PDPs

allow.pdp	Always returns a positive result.
deny.pdp	Always returns a negative result.
simplelist.pdp	Renders a positive result in case the identity corresponds to the listed ones and additionally maps the identity to an user of the workstation.
arc.pdp	Evaluates the result based on a policy XML file.
delegate.pdp	Similar to *arc.pdp* a policy file will be evaluated. But here, the policy document is extracted and transfered by delegation.collector.
pdpservice.invoker	Establishes a connection to special service in order to get a policy decision. The PDP service implements the same functionality as ARC PDP, except that the evaluation request and response is carried by SOAP messages. The benefit of the PDP Service and the PDP Service Invoker is that the policy evaluation engine can be accessed remotely and maintained centrally.

6.1 Service

Listing 6.2 contains the source code of the Secure Echo Web Service. As one can see, the only difference, compared to the source code of the previous section, is the additional code to process the *SecHandler* starting at line 92. Obviously only the incoming queue is processed and in case the *SecHandler* fails, the SOAP service returns a fault message. The file, which is revealing a lot more information, is the server configuration file which is shown in Listing 6.1. To enable *SecHandlers* to work with identities, it is almost essential to utilise the TLS layer. The TLS layer extracts the identity such that it later can be used to process the *SecHandler*. In line 23 and line 24 two additional plugins are named to be loaded which are implementing of the *SecHandlers* and *PDPs*. The first *SecHandler* is declared at line 39. The PDP *allow.pdp* always returns a positive result, such that everyone who passes the *SecHandler* is mapped to the user *nobody*. The second *SecHandler* limits the access to the SOAP services using the *arc.authz SecHandler* and the *simplelist* policy, see line 45. A similar example was already shown in Listing 6.1. The *simplelist* policy returns a positive result if the identity of the certificate corresponds to an identity within the file *clientlist.txt*. A small example of such a file is displayed in Listing 6.3.

Listing 6.2: Source code of the Secure Echo Service.

```
1 #include <arc/loader/Plugin.h>
2 #include <arc/message/PayloadSOAP.h>
3
4 #include <stdio.h>
5
6 #include "secechoservice.h"
```

```
 7
 8  /**
 9   * Method which is capable to process the argument PluginArgument.
10   * Initializes the service and returns it.
11   * It is needed to load the class as a plugin.
12   */
13  static Arc::Plugin* get_service(Arc::PluginArgument* arg)
14  {
15      Arc::ServicePluginArgument* mccarg = dynamic_cast<Arc::ServicePluginArgument*>(arg);
16      if(!mccarg) return NULL;
17      return new ArcService::SecEchoService((Arc::Config*)(*mccarg));
18  }
19
20  /**
21   * This variable is defining basic entities of the implemented service.
22   * It is needed to get the correct entry point into the plugin.
23   * @see Plugin#get_instance()
24   */
25  Arc::PluginDescriptor PLUGINS_TABLE_NAME[] = {
26      {
27          "sececho",              /* Unique name of plugin in scope of its kind   */
28          "HED:SERVICE",          /* Type/kind of plugin                          */
29          1,                      /* Version of plugin (0 if not applicable)      */
30          &get_service            /* Pointer to constructor function              */
31      },
32      { NULL, NULL, 0, NULL }     /* The array is terminated by element           */
33  };                              /* with all components set to NULL              */
34
35
36
37  using namespace Arc;
38
39  namespace ArcService
40  {
41
42      /**
43      * Constructor. Calls the super constructor, initializing the logger and
44      * setting the namespace of the payload. Furthermore the prefix and suffix will
45      * extracted out of the Config object (Elements were declared inside the HED
46      * configuration file).
47      */
48      SecEchoService::SecEchoService(Arc::Config *cfg) : Service(cfg),logger(Logger::
            rootLogger, "Echo")
49      {
50          ns_["sececho"]="urn:sececho";
51          prefix_=(std::string)((*cfg)["prefix"]);
52          suffix_=(std::string)((*cfg)["suffix"]);
53      }
54
55      /**
56      * Deconstructor. Nothing to be done here.
57      */
58      SecEchoService::~SecEchoService(void)
59      {
60      }
61
62      /**
63      * Method which creates a fault payload
64      */
65      Arc::MCC_Status SecEchoService::makeFault(Arc::Message& outmsg, const std::string &
            reason)
66      {
67          logger.msg(Arc::WARNING, "Creating fault! Reason: \"%s\"",reason);
68          Arc::PayloadSOAP* outpayload = new Arc::PayloadSOAP(ns_,true);
69          Arc::SOAPFault* fault = outpayload->Fault();
70          if(fault) {
71              fault->Code(Arc::SOAPFault::Sender);
72              fault->Reason(reason);
73          };
74          outmsg.Payload(outpayload);
75          return Arc::MCC_Status(Arc::STATUS_OK);
76      }
77
78      /**
79      * Processes the incoming message and generates an outgoing message.
80      * Returns the incoming string ordinary or reverse - depending on the operation
81      * requested.
82      * @param inmsg incoming message
83      * @param inmsg outgoing message
84      * @return Status of the result achieved
85      */
86      Arc::MCC_Status SecEchoService::process(Arc::Message& inmsg, Arc::Message& outmsg)
87      {
88          logger.msg(Arc::DEBUG, "Secure Echo Web Service has been started...");
89
90
91
```

```
92          /** Processes the SecHandlers of the service */
93          logger.msg(Arc::DEBUG, "Secure Echo Web Service: Current Client has the DN: \"%s\"",
                inmsg.Attributes()->get("TLS:PEERDN"));
94
95          if(!ProcessSecHandlers(inmsg,"incoming")) {
96              logger.msg(Arc::DEBUG, "Secure Echo Web Service: Didn't passes SecHandler!");
97              return makeFault(outmsg, "Access to this service is not permitted!");
98          }else{
99              logger.msg(Arc::DEBUG, "Secure Echo Web Service: Passed SecHandler!");
100         }
101         /** */
102
103
104
105
106         /** Both input and output are supposed to be SOAP */
107         Arc::PayloadSOAP* inpayload  = NULL;
108         Arc::PayloadSOAP* outpayload = NULL;
109         /** */
110
111         /**  Extracting incoming payload */
112         try {
113             inpayload = dynamic_cast<Arc::PayloadSOAP*>(inmsg.Payload());
114         } catch(std::exception& e) { };
115         if(!inpayload) {
116             logger.msg(Arc::ERROR, "Input is not SOAP");
117             return makeFault(outmsg, "Received message was not a valid SOAP message.");
118         };
119         /** */
120
121         /** Analyzing and execute request */
122         Arc::XMLNode requestNode  = (*inpayload)["sececho:secechoRequest"];
123         Arc::XMLNode sayNode      = requestNode["sececho:say"];
124         std::string operation = (std::string) sayNode.Attribute("operation");
125         std::string say       = (std::string) sayNode;
126         std::string hear      = "";
127         logger.msg(Arc::DEBUG, "Say: \"%s\"  Operation: \"%s\"",say,operation);
128
129         //Compare strings with constants defined in the header file
130         if(operation.compare(ECHO_TYPE_ORDINARY) == 0)
131         {
132             hear = prefix_+say+suffix_;
133         }
134         else if(operation.compare(ECHO_TYPE_REVERSE) == 0)
135         {
136             int len = say.length ();
137             int n;
138             std::string reverse = say;
139             for (n = 0;n < len; n++)
140             {
141                 reverse[len - n - 1] = say[n];
142             }
143             hear = prefix_+ reverse +suffix_;
144         }
145         else
146         {
147             return makeFault(outmsg, "Unknown operation. Please use \"ordinary\" or \"
                    reverse\"");
148         }
149         /** */
150
151         outpayload = new Arc::PayloadSOAP(ns_);
152         outpayload->NewChild("sececho:secechoResponse").NewChild("sececho:hear")=hear;
153
154         outmsg.Payload(outpayload);
155         {
156             std::string str;
157             outpayload->GetDoc(str, true);
158             logger.msg(Arc::DEBUG, "process: response=%s",str);
159         };
160
161         logger.msg(Arc::DEBUG, "Secure Echo Web Service  done...");
162         return Arc::MCC_Status(Arc::STATUS_OK);
163     }
164
165
166 }//namespace
```

The first part of an entry needs to be the distinguished name (DN) of the current client. A DN is a string which represents a user uniquely and is created using the identity of the certificate. To get the correct DN of a client one can use the TLS Echo Web Service and check the log-file. The second part of an entry specifies the user the identity shall be mapped to. In the present case, the identity will be mapped to *griduser*. In case the policy is unable to match the identity, the PDP returns a negativ result. The *SecHandler* within the

SOAP component limits the access of SOAP to a certain group of users. The last *SecHandler*, used in the server configuration file, is to be seen starting with line 61. The *SecHandler* is chosen to be *arc.authz* which leads to the fact that the access to the Echo Web Service will be limited one more time to a subset of the group. The policy *arc.pdp* returns a positive result in case the rules encapsulated within the file *policy.xml* will return a positive result.

If the PDP *arc.authz* is used, a bunch of requests will be generated for the client. The requests are compared with the rules defined in the policy and for each rule a decision is made. A sample policy is shown in Listing 6.4. The main element is *policy* which encloses three elements called *Rule*[*]. Two types of rules are possible: a permitting or a denying rule. A Rule may have four different decision states:

- PERMIT — A permitting rule returns the decision permit if one request fits to the rule and the request is fullfilled by the rule. (A denying rule would return deny in that case)
- DENY — A permitting rule returns the decision deny if one request fits to the rule and the request is not fullfilled by the rule. (A denying rule would return permit that case)
- INDETERMINATE — Requests and rule have some parts which not fits (i.e. shibboleth identity is checked, but user has identified himself via X.509 and additionally an existing resource is requested)
- NOT_APPLICABLE — If the request doesn't fit with the rule at all. (i.e. if the request is for the HTTP method GET, but the rule checks the method POST)

Once all rules are evaluated, a decision for the policy has to be generated which will depend on the results of the rules. The policy element attribute *CombiningAlg* determines how the final result will be generated. Two possibilities are given:

- **Deny-Overrides** (default)
  - If there is at least one DENY in results final result is DENY.
  - Otherwise if there is at least one PERMIT, the final result is PERMIT.
  - Otherwise if there is at least one NOT_APPLICABLE final result is NOT_APPLICABLE.
  - Otherwise final result is INDETERMINATE.
- **Permit-Overrides**
  - If there is at least one PERMIT in results final result is PERMIT.
  - Otherwise if there is at least one DENY the final result is DENY.
  - Otherwise if there is at least one NOT_APPLICABLE final result is NOT_APPLICABLE.
  - Otherwise final result is INDETERMINATE.

A rule is devided into four sub-elements: *Subjects*, *Resources*, *Actions* and *Conditions*. The element *Subjects* defines who shall be affected by that rule. The identities can be specified using the DN of the certificate. The second element determines the *Resources* which shall be examined by the rule (i.e. using a certain HTTP path). The *Actions* are specifing the intended usage of the resource the rule shall survey. The last element *Condition* defines somewhat like Time, Duration, namespace of a SOAP message etc. which are needed to be fulfilled. The detailed description of these conditions is rather extensive and beyond the scope of this tutorial. Please refer to the further going tutorials which are listed in the appendix of this tutorial.

Within the sample policy, which was shown in Listing 6.4, three different examples of rule states are prepared: PERMIT, INDETERMINATE and NOT APPLICABLE. The first rule, starting at line 4, is fitting to the server configuration file such that a user with a suitable identity will produce the state PERMIT. Since there is no rule within the policy which will have the DENY state, in particular no rule that could have matched earlier, this rule lets the client pass the PDP. The second rule begins after line 29. It is arranged in a manner such that the state INDETERMINATE will be induced. Two things are checked within it: the

[*]The latest XML schema for defining an *arc.pdp* policy can be found at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/arcpdp/Policy.xsd

```
<?xml version="1.0"?>
<ArcConfig
  xmlns="http://www.nordugrid.org/schemas/ArcConfig/2007"
  xmlns:tcp="http://www.nordugrid.org/schemas/ArcMCCTCP/2007"
  xmlns:tls="http://www.nordugrid.org/schemas/ArcMCCTLS/2007"
  xmlns:ut="http://www.nordugrid.org/schemas/UsernameTokenSH"
  xmlns:arcpdp="http://www.nordugrid.org/schemas/ArcPDP"
  xmlns:sececho="urn:sececho_config"
>
    <Server>
    <Pidfile>/var/run/arched.pid</Pidfile>
    <Logger level="VERBOSE">/var/log/arched.log</Logger>
    </Server>
    <ModuleManager>
        <Path>/tmp/arc/tutorial/lib/</Path>
        <Path>/usr/lib/arc/</Path>
    </ModuleManager>
    <Plugins><Name>mcctcp</Name></Plugins>
    <Plugins><Name>mcctls</Name></Plugins>
    <Plugins><Name>mcchttp</Name></Plugins>
    <Plugins><Name>mccsoap</Name></Plugins>
    <!-- Load PDPs and Security Handlers and identity.map-->
    <Plugins><Name>arcshc</Name></Plugins>
    <Plugins><Name>identitymap</Name></Plugins>
    <Plugins><Name>secechoservice</Name></Plugins>
    <Chain>
        <Component name="tcp.service" id="tcp">
            <next id="tls"/>
            <tcp:Listen><tcp:Port>60000</tcp:Port></tcp:Listen>
        </Component>
        <Component name="tls.service" id="tls">
            <next id="http"/>
            <KeyPath>./serviceKey.pem</KeyPath>
            <CertificatePath>./serviceCert.pem</CertificatePath>
            <CACertificatePath>./clientCA.pem</CACertificatePath>
        </Component>
        <Component name="http.service" id="http">
            <next id="soap">POST</next>
            <SecHandler name="identity.map" id="map" event="incoming">
                <PDP name="allow.pdp"><LocalName>nobody</LocalName></PDP>
            </SecHandler>
        </Component>
        <Component name="soap.service" id="soap">
            <next id="plexer"/>
            <SecHandler name="arc.authz" id="auth" event="incoming">
                <PDP name="simplelist.pdp" location="./clientlist.txt"/>
            </SecHandler>
        </Component>
        <Plexer name="plexer.service" id="plexer">
            <next id="sececho">^/sececho$</next>
        </Plexer>
        <Service name="sececho" id="sececho">
            <next id="sececho"/>
            <sececho:prefix>[ </sececho:prefix>
            <sececho:suffix> ]</sececho:suffix>
            <!-- SecHandler below calls specified Policy Decision Point components.
                    In this example only one PDP is defined - simplelist.pdp. This
                    PDP compares Distinguished Name of connecting client against
                    list of allowed DNs. DNs are stored in external file one per line.
                    They may be enclosed in '"'. -->
            <SecHandler  name="arc.authz" id="auth" event="incoming">
                <PDP name="arc.pdp">
                <PolicyStore>
                    <Location type="file">policy.xml</Location>
                </PolicyStore>
                </PDP>
            </SecHandler>
        </Service>
    </Chain>
</ArcConfig>
```

Listing 6.3: Appearence of the file used by *simplelist.pdp*.

```
$ cat clientlist.txt
"/C=US/S=Maine/OU=Literary character/O=Ka-Tet Corp./CN=Roland Deschain" griduser
"/C=US/S=Maine/OU=Literary character/O=Ka-Tet Corp./CN=Susannah Dean" griduser
$
```

identity based on authentication method Shibboleth[†] and the path of the requested HTTP URL. Due to the fact, that the HTTP path is fitting to the requested one, but the authentication was not done by Shibboleth, the rule is not able to decide for a DENY or a PERMIT state — the rule produces an INDETERMINATE state. The last rule is defined in line 46. It is constructed such that is permits the access to the HTTP method GET. One can see, that within the server configuration file the Web Service uses the method POST. If a client wants to access the Service and reaches the PDP, a couple of requests are generated which are describing the manner the client wants to access the serivice. Some of these request will concern the HTTP method GET. In these cases, the rule returns NOT APPLICABLE. In the cases where the requests for a HTTP method is missing (i.e. in the request which only concerns the SOAP access) the state of the rule will be INDETERMINATE.

To get an impression in how the requests look like, one can check the log-file (the log level needs to be set to VERBOSE).

6.2 Client

The security introduced to the server has no effect on the client, such that the source code is completely identical. Even the client configuration file doesn't need to be changed.

[†]Shibboleth is an open-source implementation for authentication and authorization based on the XML standard SAML (Security Assertion Markup Language).

Listing 6.4: Policy file for the *arc.pdp* policy decision point.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="http://www.nordugrid.org/schemas/policy-arc" PolicyId="sm-example:arcpdppolicy" CombiningAlg="Deny-Overrides">

    <Rule Effect="Permit">
        <Description>
            This rule permits the subject OU=Mitarbeiter (identity was proven by CA O=) to use the SOAP service sececho.
            This rule will end up with the state PERMIT.
        </Description>
        <Subjects>
            <Subject>
              <!--which CA is accepted -->
              <Attribute AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/tls/ca"
                Type="string">/C=US/ST=Maine/O=Ka-Tet Corp./OU=Real character/CN=Stephen King</Attribute>
              <!-- which Identity is accepted -->
              <Attribute AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/tls/identity"
                Type="string">/C=US/S=Maine/O=Ka-Tet Corp./OU=Literary character/CN=Roland Deschain</Attribute>
            </Subject>
        </Subjects>
        <Resources>
            <Resource Type="string" AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/http/path">/sececho</Resource>
        </Resources>
        <Actions>
            <Action Type="string" AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/soap/operation">secechoRequest</Action>
        </Actions>
        <Conditions/>
    </Rule>


    <Rule Effect="Permit">
        <Description>
            This rule permits the user who has authenticated himself using the Shibboleth precedure to use the HTTP path sececho.
            For the example does not use that method to sign on, the state of the rule will be INDETERMINATE. Due to non request
            having a ShibName attribute.
          </Description>
        <Subjects>
            <Subject>
                <Attribute Type="ShibName">urn:mace:shibboleth:examples</Attribute>
            </Subject>
        </Subjects>
        <Resources>
            <Resource AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/http/path" Type="string">/sececho</Resource>
        </Resources>
    </Rule>


    <Rule Effect="Permit">
        <Description>
            This rule permits the access via the HTTP method GET.
            For the example uses POST not GET. Request containing the HTTP method GET are getting the state NOT APPLICABLE.
            Request without the HTTP method end up with INDETERMINATE.
        </Description>
        <Actions>
            <Action AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/http/method" Type="string">GET</Action>
        </Actions>
    </Rule>

</Policy>
```

7 Appendix

7.1 Useful tutorials and documentations

In order to learn more about ARC several other tutorials and documentations have been written:

- ARC Web Services Quick Usage Guide [?]
- The Hosting Environment of the Advanced Resource Connector middleware [?].
- Security framework of ARC1 [?].
- Documentation of the ARC storage system [?].
- Eclipse WTP 1.5.1, Introduction to the WSDL Editor, http://wiki.eclipse.org/index.php/Introduction_to_the_WSDL_Editor

7.2 Important XSD files

Listing 7.1: XML schema of configuration files.

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

    <xsd:complexType name="Plugins_Type">
        <!--
             This element defines shared library which contains plugins to be used.
             It is supposed to be used if name of library is not same as name of plugin
             and hence can't be located automatically.
        -->
        <xsd:sequence>
            <xsd:element name="Name" type="xsd:string"/>
            <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="
                unbounded"/>
        </xsd:sequence>
        <xsd:anyAttribute namespace="##other" processContents="lax"/>
    </xsd:complexType>
    <xsd:element name="Plugins" type="Plugins_Type"/>

    <xsd:complexType name="next_Type">
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="id" type="xsd:string" use="optional"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
    <xsd:element name="next" type="next_Type"/>

    <xsd:complexType name="SecHandler_Type">
        <!--
             This element specifies security handler plugin to be called
            at various stages of message processing. Depending on produced
            result message may be either sent farther through chain or
            processing would be cancelled. Required attribute 'name' specifies
            name of plugin. Attibute 'id' creates identifier of SecHandler
            which may be used to refer to it. If attribute 'refid' is defined
            then configuration of SecHendler is provided by another element
            within ArcConfig with corresponding 'id'. Attribute 'event' defines
            to which queue inside MCC SecHandler to be attached. If it's
            missing SecHandler is attached to default queue if MCC has such.
            Names of queues are MCC specific. If not otherwise specified they
            are 'incoming' and 'outgoing' and are processed for incoming
            and outgoing messages. There is no default queue by default.
        -->
        <xsd:sequence>
```

```
49              <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="
                    unbounded"/>
50          </xsd:sequence>
51          <xsd:attribute name="name" type="xsd:string" use="required"/>
52          <xsd:attribute name="id" type="xsd:string" use="optional"/>
53          <xsd:attribute name="refid" type="xsd:string" use="optional"/>
54          <xsd:attribute name="event" type="xsd:string" use="optional"/>
55          <xsd:anyAttribute namespace="##other" processContents="lax"/>
56      </xsd:complexType>
57      <xsd:element name="SecHandler" type="SecHandler_Type"/>
58
59      <xsd:complexType name="Component_Type">
60          <!--
61               This element defines MCC plugin. Required attribute 'name' specifies name of
62              plugin as defined in MCC description. Optional attribute 'id' assigns identifier
63              which is used to refer to this element from others. If not specified it will be
64              assigned automatically. Sub-elements 'next' refer to next components in a chain
65              through their attribute 'id' and their content represent assigned component-
                    specific
66              label. If attribute 'id' is missing all 'next' refer to next component in
                    document.
67              If 'next' is missing one label-less 'next' is assigned automatically. Presence
                    of
68              attribute 'entry' exposes this MCC through Loader class interface to external
                    code.
69              That is meant to be used in code which cretes chains dynamically like client
70              utilities. Rest elements define component-specific configuration.
71          -->
72          <xsd:sequence>
73              <xsd:element ref="next"       minOccurs="0" maxOccurs="unbounded"/>
74              <xsd:element ref="SecHandler" minOccurs="0" maxOccurs="unbounded"/>
75              <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="
                    unbounded"/>
76          </xsd:sequence>
77          <xsd:attribute name="name" type="xsd:string" use="required"/>
78          <xsd:attribute name="id" type="xsd:string" use="optional"/>
79          <xsd:attribute name="entry" type="xsd:string" use="optional"/>
80          <xsd:anyAttribute namespace="##other" processContents="lax"/>
81      </xsd:complexType>
82      <xsd:element name="Component" type="Component_Type"/>
83
84      <xsd:complexType name="Plexer_Type">
85          <xsd:sequence>
86          <!--
87               This element is a Plexer. Optional attribute 'id' assigns identifier
88              which is used to refer to this element from others. If not specified it will be
89              assigned automatically. Sub-elements 'next' refer to next components in a chain
                    and
90              their content represent requested endpoints.
91              In Plexer content of element 'next' represents Regular Expression pattern.
92              For every incoming message path part of message's endpoint is matched pattern.
93              In case of ordinary service element 'next' may look like
94                <next>^/service$</next>
95              If service is also responsible for whole subtree then simple solution is
96                <next>^/service</next>
97              But more safer solution would be to use 2 elements
98                <next>^/service$</next>
99                <next>^/service/</next>
100             Unmatched part of message endpoint is propagated with incoming message in
101             attribute PLEXER:EXTENSION and may be used by service to determine response.
102         -->
103             <xsd:element ref="next"     minOccurs="0" maxOccurs="unbounded"/>
104             <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="
                    unbounded"/>
105         </xsd:sequence>
106         <xsd:attribute name="id" type="xsd:string" use="optional"/>
107         <xsd:anyAttribute namespace="##other" processContents="lax"/>
108     </xsd:complexType>
109     <xsd:element name="Plexer" type="Plexer_Type"/>
110
111     <xsd:complexType name="Service_Type">
112         <!--
113              This element represents a service - last compnent in a chain. Required
                    attribute 'name'
114             specifies name of plugin as defined in Service description. Optional attribute '
                    id'
115             assigns identifier which is used to refer to this element from others. If not
                    specified
116             it will be assigned automatically. Rest elements define service-specific
                    configuration.
117         -->
118         <xsd:sequence>
119             <xsd:element ref="SecHandler" minOccurs="0" maxOccurs="unbounded"/>
120             <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="
                    unbounded"/>
121         </xsd:sequence>
122         <xsd:attribute name="name" type="xsd:string" use="required"/>
```

```
123         <xsd:attribute name="id" type="xsd:string" use="optional"/>
124         <xsd:anyAttribute namespace="##other" processContents="lax"/>
125     </xsd:complexType>
126     <xsd:element name="Service" type="Service_Type"/>
127
128     <xsd:complexType name="Chain_Type">
129         <!--
130              This element is not required and does not affect chains directly. It's purpose
131             is to group multiple components logically mostly for readability purpose.
132         -->
133         <xsd:sequence>
134             <xsd:element ref="Chain"      minOccurs="0" maxOccurs="unbounded"/>
135             <xsd:element ref="Component"  minOccurs="0" maxOccurs="unbounded"/>
136             <xsd:element ref="Plexer"     minOccurs="0" maxOccurs="unbounded"/>
137             <xsd:element ref="Service"    minOccurs="0" maxOccurs="unbounded"/>
138             <xsd:element ref="SecHandler" minOccurs="0" maxOccurs="unbounded"/>
139         </xsd:sequence>
140     </xsd:complexType>
141     <xsd:element name="Chain" type="Chain_Type"/>
142
143     <xsd:complexType name="ModuleManager_Type">
144         <!--
145              This element specifies parameters needed to successfully load plugins.
146         Currently it allows to specify filesystem paths to directories where
147         plugin libraries are located.
148         -->
149         <xsd:sequence>
150             <xsd:element name="Path"     minOccurs="0" maxOccurs="unbounded"/>
151         </xsd:sequence>
152     </xsd:complexType>
153     <xsd:element name="ModuleManager" type="ModuleManager_Type"/>
154
155     <xsd:complexType name="ArcConfig_Type">
156         <!--
157              This is a top level element of ARC configuration document.
158         -->
159         <xsd:sequence>
160             <xsd:element ref="Chain"         minOccurs="0" maxOccurs="unbounded"/>
161             <xsd:element ref="Component"     minOccurs="0" maxOccurs="unbounded"/>
162             <xsd:element ref="Plexer"        minOccurs="0" maxOccurs="unbounded"/>
163             <xsd:element ref="Service"       minOccurs="0" maxOccurs="unbounded"/>
164             <xsd:element ref="ModuleManager" minOccurs="0" maxOccurs="1"/>
165             <xsd:element ref="SecHandler"    minOccurs="0" maxOccurs="unbounded"/>
166             <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="
                    unbounded"/>
167         </xsd:sequence>
168         <xsd:anyAttribute namespace="##other" processContents="lax"/>
169     </xsd:complexType>
170     <xsd:element name="ArcConfig" type="ArcConfig_Type"/>
171 </xsd:schema>
```

Listing 7.2: XML schema of the TCP component.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xsd:schema
 3   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 4   xmlns="http://www.nordugrid.org/schemas/ArcMCCTCP/2007"
 5   xmlns:arc="http://www.nordugrid.org/schemas/ArcConfig/2007"
 6   targetNamespace="http://www.nordugrid.org/schemas/ArcMCCTCP/2007"
 7   elementFormDefault="qualified">
 8
 9     <xsd:simpleType name="Version_Type">
10         <!-- This element defines TCP/IP protocol version.  -->
11         <xsd:restriction base="xsd:string">
12             <xsd:enumeration value="4"/>
13             <xsd:enumeration value="6"/>
14         </xsd:restriction>
15     </xsd:simpleType>
16     <xsd:element name="Version" type="Version_Type"/>
17
18     <xsd:complexType name="Listen_Type">
19         <!--
20             This element defines listening TCP socket. If interface is missing socket
21             is bound to all local interfaces (not supported). There may be multiple Listen
                   elements.
22         -->
23         <xsd:sequence>
24             <xsd:element name="Interface" type="xsd:string" minOccurs="0" maxOccurs="1"/>
25             <xsd:element name="Port" type="xsd:int" minOccurs="1" maxOccurs="1"/>
26             <xsd:element name="Version" type="Version_Type" minOccurs="0" maxOccurs="1"/>
27         </xsd:sequence>
28     </xsd:complexType>
29     <xsd:element name="Listen" type="Listen_Type"/>
30
31     <xsd:complexType name="Connect_Type">
```

```
32          <!--
33              This element defines TCP connection to be established to specified Host at
                    specified Port.
34              If LocalPort is defined TCP socket will be bound to this port number (not
                    supported).
35          -->
36          <xsd:sequence>
37              <xsd:element name="Host" type="xsd:string" minOccurs="1" maxOccurs="1"/>
38              <xsd:element name="Port" type="xsd:int" minOccurs="1" maxOccurs="1"/>
39              <xsd:element name="LocalPort" type="xsd:int" minOccurs="0" maxOccurs="1"/>
40          </xsd:sequence>
41      </xsd:complexType>
42      <xsd:element name="Connect" type="Connect_Type"/>
43
44  </xsd:schema>
```

Listing 7.3: XML schema of the TLS component.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <xsd:schema
3     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4     xmlns="http://www.nordugrid.org/schemas/ArcMCCTLS/2007"
5     xmlns:arc="http://www.nordugrid.org/schemas/ArcConfig/2007"
6     targetNamespace="http://www.nordugrid.org/schemas/ArcMCCTLS/2007"
7     elementFormDefault="qualified">
8
9       <!-- Location of private key.
10        Default is /etc/grid-security/hostkey.pem for service
11        and none for client. -->
12      <xsd:element name="KeyPath" type="xsd:string"/>
13
14      <!-- Content of private key - not supported -->
15      <xsd:element name="Key" type="xsd:string"/>
16
17      <!-- Location of public certificate.
18        Default is /etc/grid-security/hostcert.pem for service
19        and none for client. -->
20      <xsd:element name="CertificatePath" type="xsd:string"/>
21
22      <!-- Content of public certificate - not supported -->
23      <xsd:element name="Certificate" type="xsd:string"/>
24
25      <!-- Location of proxy credentials - includes certificates, key and
26        chain of involved certificates. Overwrites elements Key, KeyPath,
27        Certificate and CertificatePath.
28        Default is none for client and none for service. -->
29      <xsd:element name="ProxyPath" type="xsd:string"/>
30
31      <!-- Content of proxy credentials - not supported -->
32      <xsd:element name="Proxy" type="xsd:string"/>
33
34      <!-- Location of certificate of CA. Default is none. -->
35      <xsd:element name="CACertificatePath" type="xsd:string"/>
36      <xsd:complexType name="CACertificatesDir_Type">
37        <xsd:simpleContent>
38          <xsd:extension base="xsd:string">
39            <xsd:attribute name="PolicyGlobus" type="xsd:boolean" use="optional" default="
                  false"/>
40          </xsd:extension>
41        </xsd:simpleContent>
42      </xsd:complexType>
43
44      <!-- Content of certificate of CA - not supported -->
45      <xsd:element name="CACertificate" type="xsd:string"/>
46
47      <!-- Directory containing certificates of accepted CAs.
48        Default is /etc/grid-security/ . -->
49      <xsd:element name="CACertificatesDir" type="arc:CACertificatesDir_Type"/>
50
51      <!-- Whether checking client certificate. Only needed for service side.
52        Default is "true" . -->
53      <xsd:element name="ClientAuthn" type="xsd:string"/>
54
55      <!-- The DN list of the trusted voms server credential; in the AC part of voms
56      proxy certificate, voms proxy certificate comes with the server certificate which
57      is used to sign the AC. So when verifying the AC on the AC-consuming side (in ARC1,
58      it is the MCCTLS which will consumes the AC), the server certificate will be checked
59      against a trusted DN list. Only if the DN and issuer's DN of server certificate
60      exactly matches the DN list in the configuration under TLS component, the AC can be
61      trusted -->
62      <xsd:element name="VOMSCertTrustDNChain" type="arc:VOMSCertTrustDNChain_Type"/>
63      <xsd:complexType name="VOMSCertTrustDNChain_Type">
64          <xsd:sequence>
65            <xsd:element name="VOMSCertTrustDN" type="xsd:string"/>
66            <xsd:element name="VOMSCertTrustRegex" type="xsd:string"/>
```

```
67          </xsd:sequence>
68      </xsd:complexType>
69      <!-- DN list in an external file, which is in the same format as VOMSCertTrustDNChain
            -->
70      <xsd:element name="VOMSCertTrustDNChainsLocation" type="xsd:string"/>
71
72      <!-- Type of handshake applied. Default is TLS. -->
73      <xsd:element name="Handshake" type="arc:Handshake_Type">
74      <xsd:simpleType name="Handshake_Type">
75        <xsd:restriction base="xsd:string">
76          <xsd:enumeration value="TLS"/>
77          <xsd:enumeration value="SSLv3"/>
78        </xsd:restriction>
79      </xsd:simpleType>
80
81  </xsd:schema>
```

Listing 7.4: XML schema of the HTTP component.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <xsd:schema
3     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4     xmlns="http://www.nordugrid.org/schemas/ArcMCCHTTP/2007"
5     xmlns:arc="http://www.nordugrid.org/schemas/ArcMCCHTTP/2007"
6     targetNamespace="http://www.nordugrid.org/schemas/ArcMCCHTTP/2007"
7     elementFormDefault="qualified">
8
9       <!--
10          These elements define configuration parameters for client
11          part of HTTP MCC. Service part uses no configuration.
12      -->
13      <!--
14          Endpoint (URL) for HTTP request. This configuration parameter
15         may be overwritted by HTTP:ENDPOINT attribute of message.
16      -->
17      <xsd:element name="Endpoint" type="xsd:string"/>
18      <!--
19         Method to use for HTTP request. May be overwritten by
20        HTTP:METHOD attribute of message.
21      -->
22      <xsd:element name="Method" type="xsd:string"/>
23  </xsd:schema>
```