Knowarc_D0.0-0_00
14/12/2017

The SWIG wrapped ARC Python API and the arcom utility package

Tamás Kazinczy[1]

[1] kazy@niif.hu

Contents

1 Preface

When it comes to creating new functionality in ARC[2] it is nice if one could do that in a simple, quick and neat way. Python[3] is known for its versatility, simple and clean design, that made it a good choice for rapid prototyping and gluing. As the new ARC has moved towards the service oriented architecture, it has become a viable option to use Python both for creating new services and gluing existing ones together. To be able to work with the functionality provided by ARC HED[4], a decision was made to wrap its API with SWIG[5]. Furthermore, the arcom utility package was created on the basis of Python modules that had been created to ease developers' lives. (It contains reusable components to support various parts of development, including but not limited to the following categories: client development, logging, security and service development.)

1.1 Purpose of this document

Objectives of this document are:

- to demonstrate differences of original and wrapped API
- to comment about the use of ARC Python API
- to document the arcom utility package

1.2 Structure

The document has the following structure:

- In the first part we provide an overview of how the API is wrapped by SWIG. Here the use of SWIG interface files will be discussed along with the main differences resulting from them and some thoughts will be given about usage.
- In the second part the arcom utility package will be discussed. Here the functionality found in the package will be described and some examples of use will be shown.

2 The SWIG generated API

In this part the following topics will be covered:

- Python specific parts of SWIG interface files
- main differences between the original and the SWIG generated API resulting from the style of wrapping
- the use of the generated API from Python

2.1 Python specific parts of SWIG interface files

SWIG interface files are found in the swig subdirectory under the ARC source directory.

Specific parts for Python in SWIG interface files are enclosed in SWIGPYTHON blocks, starting with *#ifdef SWIGPYTHON* and ending with *#endif*.

There are seven interface files that contain such blocks:

- Arc.i

[2] Advanced Resource Connector; `http://www.nordugrid.org/middleware/`
[3] `http://www.python.org/`
[4] Hosting Environment Daemon; `http://www.nordugrid.org/documents/ARCHED_article.pdf`
[5] Simplified Wrapper and Interface Generator; `http://www.swig.org/`

- common.i
- message.i
- client.i
- data.i
- delegation.i
- security.i

When not in a specific block (neither Python nor another - e.g. SWIGJAVA), instructions are applied to all languages.

2.1.1 Arc.i

- `%module arc`
  As stated in the SWIG documentation[6]: "The `%module` directive defines the name of the module that will be created by SWIG." That means, this module could be imported in Python code with: *import arc*

- `%include <stl.i>`
  Include the Standard Template Library, so that templates could be used.

- ```
  #ifdef SWIGPYTHON
  %include <std_list.i>
  #endif
  ```
  If target language is Python include `std_list`.

- `%template(StringList) std::list<std::string>;`
  Define StringList as a template list of strings. An example of handling a list of strings is shown below.

[6] `http://www.swig.org/Doc1.3/SWIGDocumentation.html`
  ```

Example 1 Handling a list of strings

```
>>> import arc
>>> #create an empty list
... sl = arc.StringList()
>>> #append strings to the list
... #size increases
... sl.size()
0
>>> sl.append('apple')
>>> sl.size()
1
>>> sl.append('banana')
>>> sl.size()
2
>>> sl.append('lemon')
>>> sl.size()
3
>>> sl.append('orange')
>>> sl.size()
4
>>> #list members
... sl[0], sl[1], sl[2], sl[3]
('apple', 'banana', 'lemon', 'orange')
>>> #do some slicing
... tmp = sl[:2]
>>> #now tmp contains the first two strings
... tmp.size()
2
>>> tmp[0], tmp[1]
('apple', 'banana')
>>> #do some more slicing
... tmp = sl[1:]
>>> #now tmp contains all the strings except the first
... tmp.size()
3
>>> tmp[0], tmp[1], tmp[2]
('banana', 'lemon', 'orange')
```

- `%template(StringStringMap) std::map<std::string, std::string>;`
  Define StringStringMap as a template map where both key and value are strings.

Example 2 arc.StringStringMap

```
>>> import arc
>>> #create an empty map
... ssm = arc.StringStringMap()
>>> #add mapping
... ssm['key1'] = 'value1'
>>> ssm['key2'] = 'value2'
>>> #get keys
... ssm.keys()
['key1', 'key2']
>>> #get value for 'key1'
... ssm['key1']
'value1'
>>> #get value for 'key2'
... ssm['key2']
'value2'
```

- ```
  #ifdef SWIGPYTHON
  namespace Arc {

  %typemap(in, numinputs=0) std::string& content (std::string str) {
    $1 = &str;
  }
  ```

  If target language is Python then this input typemap tells SWIG that "content" arguments of type "`std::string&`" found in "Arc" namespace are to be ignored. As stated in the SWIG documentation: "When numinputs is set to 0, the argument is effectively ignored and cannot be supplied from the target language. " However the argument is still required when making the C/C++ call. This is solved by providing a locally declared variable called str from which the value used is obtained. Statement "`$1 = &str;`" sets the input argument to point to this temporary variable.

  ```
  %typemap(argout) std::string& content {
    PyObject *tuple;
    tuple = PyTuple_New(2);
    PyTuple_SetItem(tuple,0,$result);
    PyTuple_SetItem(tuple,1,Py_BuildValue("s",$1->c_str()));
    $result = tuple;
  }
  ```

  As the SWIG documentation tells: "The `"argout"` typemap is used to return values from arguments." With the previous typemap in mind this one indicates that the result will be changed as follows:

  - A tuple will be created with two members.
  - The first member comes from the original result.
  - The second one comes from the argument.
  - The tuple will be returned as the new result.

- ```
  %include "common.i"
  %include "message.i"
  %include "client.i"
  %include "data.i"
  %include "delegation.i"
  ```

  Include other interface files.

- ```
  #ifdef SWIGPYTHON
  %include "security.i"
  #endif
  ```

  If target language is Python include interface file: security.i
  ```

2.1.2 common.i

- `%include <typemaps.i>`
  Typemaps will be used.

- `%include <std_vector.i>`
  The `std_vector` library will be used.

- ```
  %ignore operator !;
  %ignore operator [];
  %ignore operator =;
  %ignore operator ++;
  %ignore operator <<;

  %ignore *::operator [];
  %ignore *::operator ++;
  %ignore *::operator --;
  %ignore *::operator =;

  %ignore Arc::MatchXMLName;
  %ignore Arc::MatchXMLNamespace;

  %template(XMLNodeList) std::list<Arc::XMLNode>;
  %template(URLList) std::list<Arc::URL>;
  %template(URLVector) std::vector<Arc::URL>;
  %template(URLListMap) std::map< std::string, std::list<Arc::URL> >;
  ```

  As stated in SWIG documentation: "`%ignore` instructs SWIG to ignore declarations that match a given identifier. Any function, variable, etc. which matches `%ignore` will not be wrapped and therefore will not be available from the target language." According to this, ignored operators and methods above will not be accessible from Python.
  Define XMLNodeList as a template list of Arc::XMLNode and URLList as a template list of Arc::URL objects. Define URLVector as a template vector of Arc::URL objects and URLListMap as a template map where the key is a string and the value is a list of Arc::URL objects. (See Example1 for an example of using a list of objects - list of strings in that particular case - in Python.)

- ```
  %rename(toBool) operator bool;
  %rename(__str__) operator std::string;
  ```

  If target language is Python then apply the following:

  - The operator "bool" is being renamed to "toBool" and is thus accessible from Python that way. However, "toBool" does not appear in the Python API, that is, there is no trace of "toBool" either in the generated code or in error messages.
  - The operator "std::string" is being renamed to "`__str__`" and is thus accessible from Python that way.

Example 3 String operator - simple node

```
>>> import arc
>>> #create an XMLNode
... mynode = arc.XMLNode(arc.NS({'me':'http://example.com/myExample'}),'me:myNode')
>>> #set text content
... mynode.Set('Hello, World!')
>>> #representation of the node
... mynode.GetXML()
'<me:myNode xmlns:me="http://example.com/myExample">Hello, World!</me:myNode>'
>>> #String operator in action
... str(mynode)
'Hello, World!'
```

Example 4 String operator - complex node

```
>>> # String operator of XMLNode works with the text content.
... # It does not care about child nodes or text contents of those nodes.
>>>
>>> # build a tree and set text content in nodes
... #       r
... #     / | \
... #    /  |  \
... #   /   |   \
... # c01  c02  c03
... #      /\    |
... #     /  \  c06
... #   c04  c05
...
>>> # create the root node
... r = arc.XMLNode(arc.NS(),'r')
>>> # set text content
... r.Set('R')
>>> # create rest of the tree
... c01 = r.NewChild('c01')
>>> c01.Set('C-01')
>>> c02 = r.NewChild('c02')
>>> c02.Set('C-02')
>>> c03 = r.NewChild('c03')
>>> c03.Set('C-03')
>>> c04 = c02.NewChild('c04')
>>> c04.Set('C-04')
>>> c05 = c02.NewChild('c05')
>>> c05.Set('C-05')
>>> c06 = c03.NewChild('c06')
>>> c06.Set('C-06')
>>> # String operator does not care about child nodes
... str(c03)
'C-03'
>>> str(c02)
'C-02'
>>> str(r)
'R'
>>> # XML representation of a complex node
... c02.GetXML()
'<c02>C-02<c04>C-04</c04><c05>C-05</c05></c02>'
```

- `%rename(_print) Arc::Config::print;`

"Arc::Config::print" is being renamed to "`_print`" and is thus accessible from Python that way.

- ```
  %apply std::string& OUTPUT { std::string& out_xml_str };
  %include "../src/hed/libs/common/XMLNode.h"
  %clear std::string& out_xml_str;
  ```

  It is common that a function provides return values through parameters (pointers or references). In Python, such functions should have multiple return values. This typemap tells SWIG that all occurrences of "`out_xml_str`" of type "`std::string&`" as a parameter in XMLNode.h should be turned into "`std::string&`" output values instead. An example regarding the GetDoc function is shown below.

Example 5 Occurence of out_xml_str as an output value

```
>>> import arc
>>> #create node
... n = arc.XMLNode(arc.NS({'me':'myNS'}),'me:myNode')
>>> #set content
... n.Set('Hello, World!')
>>> #get document
... #GetDoc according to API:
... #void GetDoc(std::string &out_xml_str, bool user_friendly=false) const
... #out_xml_str is turned into an output value and user_friendly is an optional
... #parameter so there are no necessary parameters this time
... #result will hold the output value
... result = n.GetDoc()
>>> #show result
... result
'<?xml version="1.0"?>\n<me:myNode xmlns:me="myNS">Hello, World!</me:myNode>\n'
```

- `%rename(LogStream_ostream) LogStream;`
  LogStream is being renamed to "`LogStream_ostream`". It is so, because a function with the same name will be defined later. (See below.)

- ```
  %inline %{
  class CPyOutbuf : public std::streambuf {
   ...
  };

  class CPyOstream : public std::ostream {
   ...
  };

  %}
  ```

  Classes CPyOutBuf and CPyOstream are created. These are to be used when creating LogStreams. (Python does not have stream objects - it uses files instead - so these classes are defined to be able to represent a Python file as an ostream that is required by the C++ code.) Although they will be accessible from Python and thus could be used to create LogStreams, it is much more comfortable to take advantage of the function defined right after. (Complete source code can be found in Appendix A of this document.)

- ```
  %pythoncode %{
      def LogStream(file):
          os = CPyOstream(file)
          os.thisown = False
          ls = LogStream_ostream(os)
          ls.thisown = False
          return ls
  ```
  ```

%}

Provides an easy-to-use function to create LogStreams with.

Example 6 Using LogStream to add new destination to the root logger

```
>>> import arc
>>> import sys
>>> #get root logger
... root_logger = arc.Logger_getRootLogger()
>>> #create a LogStream; sys.stdout would be OK
... stream = arc.LogStream(sys.stdout)
>>> #add destination to root logger
... root_logger.addDestination(stream)
>>> #log a message
... #result immediately appears on sys.stdout
... root_logger.msg(arc.INFO, 'Hello, World!')
[2009-05-12 15:39:07] [Arc] [INFO] [28021/161022392] Hello, World!
```

2.1.3 message.i

- `%include <typemaps.i>`
  Typemaps will be used.

- `%rename(next) Arc::AttributeIterator::operator++;`
  Rename "operator++" of Arc::AttributeIterator to "next". For an example see the wrapping of Arc::MessageAttributes::getAll below.

- ```
  #ifdef SWIGPYTHON
  %pythonappend Arc::MessageAttributes::getAll %{
      d = dict()
      while val.hasMore():
          d[val.key()] = val.__ref__()
          val.next()
      return d
  %}
  #endif
  ```

  If target language is Python then append the above code to "Arc::MessageAttributes::getAll". Basically this means that MessageAttributes are rather returned as a dictionary. (The original result is an iterator. This iterator is used in the appended code to collect MessageAttributes and put them in a dictionary which is more natural to use in Python.)
  ```

Example 7 MessageAttributes - getAll

```
>>> import arc
>>> #create MessageAttributes object
... mas = arc.MessageAttributes()
>>> #add some key-value pairs
... mas.add('key1', 'value1')
>>> mas.add('key3', 'value3')
>>> mas.add('key2', 'value2')
>>> #get all attributes
... all = mas.getAll()
>>> #show result
... all
{'key3': 'value3', 'key2': 'value2', 'key1': 'value1'}
>>> #function returns a dictionary
... type(all)
<type 'dict'>
>>> #show keys
... all.keys()
['key3', 'key2', 'key1']
>>> #show values
... all.values()
['value3', 'value2', 'value1']
```

- ```
  %apply std::string& OUTPUT { std::string &val };
  %include "../src/hed/libs/message/SecAttr.h"
  %clear std::string &val;
  ```

  This typemap tells SWIG that all occurrences of "val" of type "`std::string&`" as a parameter in SecAttr.h should be turned into "`std::string&`" output values instead. An example could be the Export functionality of SecAttr. However, SecAttr is a base class that should be extended, and such classes are not yet available for the Python API.

- ```
  #ifdef SWIGPYTHON
  %pythonprepend Arc::MessageAuth::Export %{
        x = XMLNode("<dummy/>")
        args = args[:-1] + (args[-1].fget(), x)
  %}
  %pythonappend Arc::MessageAuth::Export %{
        return x
  %}
  #endif
  ```

  If target language is Python then "Arc::MessageAuth::Export" is modified in a way that:

  - A dummy node is created that will be replaced with the actual output.
  - The argument tuple (args) of the method is recreated to contain the dummy node mentioned above. (It will be the last element of the tuple.) Note that the fget() call is required to get the SecAttrFormat from the property object as current SecAttrFormats are defined as property objects in the SecAttr class.
  - Finally, the new output - that used to be the dummy node - is returned.

  An example would require the use of SecAttr here, therefore one is not yet available for the Python API. (An own extension of SecAttr on the Python side would not be enough here because it would not be used by MessageAuth.)

- ```
  %apply std::string& OUTPUT { std::string& out_xml_str };
  %include "../src/hed/libs/message/SOAPEnvelope.h"
  %clear std::string& out_xml_str;
  ```
  ```

This tells SWIG that all occurrences of "`out_xml_str`" of type "`std::string&`" as a parameter in SOAPEnvelope.h should be turned into "`std::string&`" output values instead. An example regarding the GetXML function is shown below.

Example 8 SOAPEnvelope - out_xml_str

```
>>> import arc
>>> #create a namespace
... ns = arc.NS({'me':'http://example.com/myExample'})
>>> #create an empty SOAPEnvelope
... #use the namespace created above
... se = arc.SOAPEnvelope(ns,False)
>>> #show it
... se.GetXML()
'<soap-env:Envelope xmlns:me="http://example.com/myExample" xmlns:soap-enc="http
://schemas.xmlsoap.org/soap/encoding/" xmlns:soap-env="http://schemas.xmlsoap.or
g/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http:/
/www.w3.org/2001/XMLSchema-instance"><soap-env:Body/></soap-env:Envelope>'
```

2.1.4 client.i

- ```
  %template(ExecutionTargetList) std::list<Arc::ExecutionTarget>;
  %template(JobControllerList) std::list<Arc::JobController *>;
  %template(JobList) std::list<Arc::Job>;
  %template(JobStateList) std::list<Arc::JobState>;
  %template(SourceTypeList) std::list<Arc::DataSourceType>;
  %template(TargetTypeList) std::list<Arc::DataTargetType>;
  %template(FileTypeList) std::list<Arc::FileType>;
  %template(DirectoryTypeList) std::list<Arc::DirectoryType>;
  %template(ApplicationEnvironmentList) std::list<Arc::ApplicationEnvironment>;
  %template(SoftwareList) std::list<Arc::Software>;
  %template(SoftwareRequirementList) std::list<Arc::SoftwareRequirement>;
  %template(ResourceTargetTypeList) std::list<Arc::ResourceTargetType>;
  ```

  Define ExecutionTargetList as a template list of Arc::ExecutionTarget objects, JobControllerList as a template list of Arc::JobController pointers, JobList as a template list of Arc::Job, JobStateList as a template list of Arc::JobState, SourceTypeList as a template list of Arc::DataSourceType, TargetTypeList as a template list of Arc::DataTargetType, FileTypeList as a template list of Arc::FileType, DirectoryTypeList as a template list of Arc::DirectoryType, ApplicationEnvironmentList as a template list of Arc::ApplicationEnvironment, SoftwareList as a template list of Arc::Software, SoftwareRequirementList as a template list of Arc::SoftwareRequirement and ResourceTargetTypeList as a template list of Arc::ResourceTargetType objects.
  (See Example1 for an example of using a list of objects in Python.)

- ```
  #ifdef SWIGPYTHON
  namespace Arc {

  %typemap(in, numinputs=0) PayloadSOAP ** response (PayloadSOAP *temp) {
      $1 = &temp;
  }
  ```

  If target language is Python then this input typemap tells SWIG that "response" arguments of type "PayloadSOAP **" found in "Arc" namespace are to be ignored. As stated in the SWIG documentation: "When numinputs is set to 0, the argument is effectively ignored and cannot be

supplied from the target language. " However the argument is still required when making the C/C++ call. This is solved by providing a locally declared variable called temp from which the value used is obtained. Statement "`$1 = &temp;`" sets the input argument to point to this temporary variable.

```
%typemap(argout) PayloadSOAP ** response {
    PyObject *o, *tuple;
    o = SWIG_NewPointerObj(*$1, SWIGTYPE_p_Arc__PayloadSOAP, SWIG_POINTER_OWN | 0 );
    tuple = PyTuple_New(2);
    PyTuple_SetItem(tuple,0,o);
    PyTuple_SetItem(tuple,1,$result);
    $result = tuple;
}
```

This argout typemap - with the previous typemap in mind - indicates that the result will be changed as follows:

- A tuple will be created with two members.
- The first member comes from the argument.
- The second one comes from the original result.
- The tuple will be returned as the new result.

When processing a SOAP request it is required to return `MCC_Status` along with the SOAP response. This is achieved by returning a tuple that contains both of them. An example of calling the Echo service is shown below.

Example 9 ClientSOAP - process

```
>>> import arc
>>> #create default config
... cfg = arc.MCCConfig()
>>> #create URL
... url = arc.URL('http://localhost:50000/Echo')
>>> #create payload
... payload = arc.PayloadSOAP(arc.NS({'echo':'urn:echo'}))
>>> #create payload content
... #and set echo message
... payload.NewChild('echo:echo').NewChild('echo:say').Set('Hello, World!')
>>> #create client
... client = arc.ClientSOAP(cfg,url)
>>> #let the client call the service
... response, status = client.process(payload)
>>> #see response
... print response.GetXML()
<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/" x
mlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XM
LSchema-instance"><soap-env:Body><echo:echoResponse><echo:hear>[ Hello, World! ]
</echo:hear></echo:echoResponse></soap-env:Body></soap-env:Envelope>
>>> #check status
... status.getKind() == arc.STATUS_OK
True
```

- `%apply std::string& INOUT { std::string& delegation_id };`
  `%include "../src/hed/libs/client/ClientX509Delegation.h"`
  `%clear std::string& delegation_id;`

This tells SWIG that all occurrences of "`delegation_id`" of type "`std::string&`" as a parameter in ClientX509Delegation.h should be used as input and turned into "`std::string&`" output values as

well. In fact, it depends on the type of delegation whether "`delegation_id`" is used as an input or is created and returned as an output value. (See example below.)

Example 10 ClientX509Delegation - delegation_id

```
For example the class ClientX509Delegation have a method createDelegation that makes
use of delegation_id:
  "For gridsite delegation service, the delegation_id
   is supposed to be created by client side, and sent to service side;
   for ARC delegation service, the delegation_id is supposed to be created
   by service side, and returned back. So for gridsite delegation service,
   this parameter is treated as input, while for ARC delegation service,
   it is treated as output."
However, this implies that one should take great care when dealing with delegation_id.
```

2.1.5 data.i

- ```
  #ifdef SWIGPYTHON
  namespace Arc {
  ```

  If target language is Python then proceed with the following rules in "Arc" namespace.

- ```
  %typemap(in, numinputs=0) std::list<FileInfo> & files (std::list<FileInfo> temp) {
    $1 = &temp;
  }
  ```

  This input typemap tells SWIG that "files" arguments of type "`std::list<FileInfo>&`" are to be ignored. However the argument is still required when making the C/C++ call. This is solved by providing a locally declared variable called temp from which the value used is obtained. Statement "`$1 = &temp;`" sets the input argument to point to this temporary variable.

  ```
  %typemap(argout) std::list<FileInfo> & files {
    PyObject *o, *tuple;
    o = PyList_New(0);
    std::list<Arc::FileInfo>::iterator it;
    for (it = (*$1).begin(); it != (*$1).end(); ++it) {
        PyList_Append(o, SWIG_NewPointerObj(new Arc::FileInfo(*it),
            SWIGTYPE_p_Arc__FileInfo, SWIG_POINTER_OWN | 0 ));
    }
    tuple = PyTuple_New(2);
    PyTuple_SetItem(tuple,0,o);
    PyTuple_SetItem(tuple,1,$result);
    $result = tuple;
  }
  ```

  This argout typemap - with the previous typemap in mind - indicates that the result will be changed as follows:

  - A tuple will be created with two members.
  - The first member comes from the argument.
  - The second one comes from the original result.
  - The tuple will be returned as the new result.

Example 11 DataPointARC - ListFiles

```
One possible use of the FileInfo list could be the ListFiles funcionality of the
DataPointARC class. This returns DataStatus along with the collected files. An example
of using ListFiles will be shown later.
```

See Example17 for details.

- `%ignore Arc::DataHandle::operator->;`
  Ignored operator will not be accessible from Python.

2.1.6 delegation.i

- ```
  #ifdef SWIGPYTHON
  %ignore Arc::DelegationConsumer::Acquire(std::string&, std::string&);
  %ignore Arc::DelegationConsumerSOAP::UpdateCredentials(std::string&, std::string&,
    const SOAPEnvelope&, SOAPEnvelope&);
  %ignore Arc::DelegationConsumerSOAP::DelegatedToken(std::string&, std::string&,
    const SOAPEnvelope&, SOAPEnvelope&);
  %ignore Arc::DelegationConsumerSOAP::DelegatedToken(std::string&, std::string&,
    const XMLNode&);
  %ignore Arc::DelegationContainerSOAP::UpdateCredentials(std::string&,
    std::string&, const SOAPEnvelope&, SOAPEnvelope&);
  %ignore Arc::DelegationContainerSOAP::DelegatedToken(std::string&, std::string&,
    const SOAPEnvelope&, SOAPEnvelope&);
  %ignore Arc::DelegationContainerSOAP::DelegatedToken(std::string&, std::string&,
    const XMLNode&);
  #endif
  ```

  If target language is Python the selected methods are to be ignored and thus will not be accessible.

- ```
  %apply std::string& OUTPUT { std::string& credentials };
  %apply std::string& OUTPUT { std::string& identity };
  %ignore DelegationContainerSOAP::UpdateCredentials(
        std::string &,const SOAPEnvelope &, SOAPEnvelope &);
  %include "../src/hed/libs/delegation/DelegationInterface.h"
  %clear std::string& identity;
  %clear std::string& credentials;
  ```

  These tell SWIG that all occurrences of "credentials" and "identity" of type "`std::string&`" in DelegationInterface.h should be turned into "`std::string&`" output values. Furthermore, "DelegationContainerSOAP::UpdateCredentials" - with the parameters above - will not be available from Python. Note that this is required by SWIG to properly generate Python code because applying the previous typemap would result in two UpdateCredentials method having exactly the same parameter list.

Example 12 DelegationConsumerSOAP - UpdateCredentials

```
UpdateCredentials functionality of DelegationConsumerSOAP contains both credentials
and identity according to the C++ API documentation:

bool Arc::DelegationConsumerSOAP::UpdateCredentials (std::string & credentials,
std::string & identity, const SOAPEnvelope & in, SOAPEnvelope & out)

Occurences are turned into output values, that is, Python API will have UpdateCredentials
take two parameters (both of them being of type SOAPEnvelope) and return a list.
```

- `%template(ScalableTimeInt) Arc::ScalableTime<int>;`

```
%template(RangeInt) Arc::Range<int>;
%template(RangeInt64) Arc::Range<int64_t>;
```

These are template instantiations for template classes `Arc::ScalableTime<int>`, `Arc::Range<int>` and `Arc::Range<int64_t>`.

2.1.7 security.i

This interface file differs from previous ones in that it does not contain implicit wrapping. That is, content will be wrapped according to explicit statements alone found in this file. Examples regarding security will be shown later in the discussion of the "arcom.security" module.

- `namespace ArcSec`
  Proceed with the following rules in "ArcSec" namespace.

- ```
  %nodefaultctor Policy;
  class Policy  {};

  %nodefaultctor Request;
  class Request {};
  ```

  No default constructor will be generated for classes Policy and Request.

- ```
  typedef enum {
    DECISION_PERMIT = 0,
    DECISION_DENY = 1,
    DECISION_INDETERMINATE = 2,
    DECISION_NOT_APPLICABLE = 3
  } Result;

  typedef struct {
    Result res;
  } ResponseItem;
  ```

  Define Result and ResponseItem.

- ```
  class ResponseList {
  public:
    int size() ;
    ResponseItem* getItem(int n);
    ResponseItem* operator[](int n);
    bool empty();
  };

  class Response {
  public:
    ResponseList& getResponseItems ();
  };

  class Source {
   public:
    Source(const Source& s):node(s.node);
    Source(Arc::XMLNode& xml);
    Source(std::istream& stream);
    Source(Arc::URL& url);
    Source(const std::string& str);
  ```
  ```

```
};

class SourceFile: public Source {
 public:
  SourceFile(const SourceFile& s):Source(s),stream(NULL);
  SourceFile(const char* name);
  SourceFile(const std::string& name);
};

class SourceURL: public Source {
 public:
  SourceURL(const SourceURL& s):Source(s),url(NULL);
  SourceURL(const char* url);
  SourceURL(const std::string& url);
};
```

These tells SWIG the classes and their methods to be wrapped.

- ```
  %nodefaultctor Evaluator;
  %newobject Evaluator::evaluate;
  class Evaluator {
  public:
      void addPolicy(const Source& policy,const std::string& id = "");
      %apply SWIGTYPE *DISOWN {Policy *policy};
      void addPolicy(Policy* policy,const std::string& id = "");
      %clear Policy *policy;
      Response* evaluate(Request* request);
      Response* evaluate(const Source& request);
      Response* evaluate(const Source& request, const Source& policy);
      Response* evaluate(const Source& request, Policy* policyobj);
      Response* evaluate(Request* request, Policy* policyobj);
      Response* evaluate(Request* request, const Source& policy);
  };

  %newobject EvaluatorLoader::getEvaluator;
  %newobject EvaluatorLoader::getRequest;
  %newobject EvaluatorLoader::getPolicy;
  class EvaluatorLoader {
  public:
      EvaluatorLoader();
      Evaluator* getEvaluator(const std::string& classname);
      Request* getRequest(const std::string& classname,
                          const Source& requestsource);
      Policy* getPolicy(const std::string& classname,
                        const Source& policysource);
  };
  ```

  No default constructor will be generated for the Evaluator class.
  SWIG is also told about additional classes to be wrapped. Furthermore, a hint is given to SWIG about ownership of objects in libarcsecurity.
  ```

3 The arcom utility package

This part is dedicated to documenting the arcom utility package. As it is meant to ease developers' lives, the arcom package comes with a handful of helper classes and utility functions. The package has the following structure:

- `__init__`
  Provides a set of readily available utility functions. These functions provide a way of e.g. importing classes or better handling of XML structures.
- client
  Provides a simple class (Client) for sending SOAP messages to services.
- logger
  Defines log levels and provides a function to get a logger of choice.
- security
  Collection of classes and functions that helps when dealing with authorization requests, policies and decision making.
- service
  Provides a base class for service definition. This base class should be extended then.
- threadpool
  Thread pool support. Consists of classes ThreadPool, ThreadPoolThread and ReadWriteLock. The latter is a lock object that allows many simultaneous "read locks", but only one "write lock."
- xmltree
  Provides the XMLTree class which provides a way to convert from XML to native Python structures and vice versa.
- store
  Provides classes for storing arbitrary objects locally.
  - basestore
    Provides a base class for stores. This base class should be extended.
  - cachedpicklestore
    Class for storing objects to files using the Python Pickle module[7]. This class keeps all the objects in memory.
  - cachedstringstore
    Class for storing objects to files as strings. This class keeps all the objects in memory.
  - picklestore
    Class for storing objects to files using the Python Pickle module.
  - stringstore
    Class for storing objects to files as strings.
  - transdbstore
    Class for storing objects to a transactional Berkeley DB[8].
  - zodbstore
    Class for storing objects to a Zope Object Database[9].

3.1 __init__.py

"`__init__.py`" contains those parts of the arcom package that are readily accessible without any further action (apart from importing the arcom package itself).

[7] `http://docs.python.org/library/pickle.html`
[8] `http://www.oracle.com/database/berkeley-db/db/index.html`
[9] `http://www.zope.org/Products/StandaloneZODB`

- `import_class_from_string(str)`
  Imports a class given as a string parameter.
  The string parameter has the format: "[package.]*module.classname", that is, package or packages (separated by periods) followed by module name and finally the class name.

Example 13 Importing the Logger class from module "logger" of "arcom" package

```
>>> import arcom
>>> # From the logger module within the arcom package
... # import the Logger class
... Logger = arcom.import_class_from_string('arcom.logger.Logger')
>>> # Now the class could be access through Logger
... Logger
<class arcom.logger.Logger at 0x89081dc>
```

- `get_attributes(node)`
  Returns all attributes of the supplied XMLNode (node) in a dictionary where attribute names will serve as keys.

Example 14 Getting attributes from an XMLNode

```
>>> import arc
>>> import arcom
>>> # Create XMLNode
... n = arc.XMLNode(arc.NS({'me':'http://example.com/myExample'}),'me:myExample')
>>> # Add attributes to node and set their values
... n.NewAttribute('foo').Set('Hello')
>>> n.NewAttribute('moo').Set('World')
>>> # Show XML
... n.GetXML()
'<me:myExample xmlns:me="http://example.com/myExample" foo="Hello" moo="World"/>'
>>> # get attributes of this XMLNode
... a = arcom.get_attributes(n)
>>> # result is a dictionary
... type(a)
<type 'dict'>
>>> # show dictionary
... a
{'foo': 'Hello', 'moo': 'World'}
>>> # extracting an attribute value
... a['moo']
'World'
```

- `get_child_nodes(node)`
  Gets children of the supplied XMLNode (node). Result will be returned as a list of XMLNodes.

  ```
  In the example below, the following structure will be created:

  <a>A
      <b>B</b>
      <c>C
          <e>E</e>
          <f>F</f>
      </c>
      <d>D
          <g>G</g>
      </d>
  <a>

  Relationships:

          a       children: (b, c, d)
         /|\
        / | \
       b  c  d    children: () (e, f) (g)
         / \  \
        e   f  g  children: () () ()
  ```

Example 15 Getting child nodes

```
>>> import arc
>>> import arcom
>>> # Create root node
... a = arc.XMLNode(arc.NS(),'a')
>>> # Create child nodes for 'a'
... b = a.NewChild('b')
>>> c = a.NewChild('c')
>>> d = a.NewChild('d')
>>> # Create child nodes for 'c'
... e = c.NewChild('e')
>>> f = c.NewChild('f')
>>> # Create child node for 'd'
... g = d.NewChild('g')
>>> # show XML
... a.GetXML()
'<a><b/><c><e/><f/></c><d><g/></d></a>'
>>> # get child nodes for 'a'
... tmp = arcom.get_child_nodes(a)
>>> # result is a list
... type(tmp)
<type 'list'>
>>> # 'a' has 3 children
... len(tmp)
3
>>> # show name for each
... ', '.join(x.Name() for x in tmp)
'b, c, d'
>>> # 'b' has no children
... tmp = arcom.get_child_nodes(b)
>>> len(tmp)
0
>>> # 'c' has 2 children: 'e' and 'f'
... tmp = arcom.get_child_nodes(c)
>>> len(tmp)
2
>>> ', '.join(x.Name() for x in tmp)
'e, f'
>>> # 'd' has 1 child: 'g'
... tmp = arcom.get_child_nodes(d)
>>> len(tmp)
1
>>> ', '.join(x.Name() for x in tmp)
'g'
>>> # nodes 'e', 'f' and 'g' have no children
... tmp = arcom.get_child_nodes(e)
>>> len(tmp)
0
>>> tmp = arcom.get_child_nodes(f)
>>> len(tmp)
0
>>> tmp = arcom.get_child_nodes(g)
>>> len(tmp)
0
```

- `get_child_values_by_name(node, name)`
  Gets values from children of a supplied XMLNode (node), where those children have the supplied

name (name).

Example 16 Get values of specified children

In the example below, the following structure will be created:

```
<node>
    <same>firstEQ</same>
    <same>secondEQ</same>
    <different>DIFF</different>
</node>
```

```
>>> import arc
>>> import arcom
>>> # Create XMLNode
... n = arc.XMLNode(arc.NS(),'node')
>>> # Create 3 child nodes (two of which get the same name)
... x = n.NewChild('same')
>>> y = n.NewChild('same')
>>> z = n.NewChild('different')
>>> # Set values for nodes
... x.Set('firstEQ')
>>> y.Set('secondEQ')
>>> z.Set('DIFF')
>>> # Show XML
... n.GetXML()
'<node><same>firstEQ</same><same>secondEQ</same><different>DIFF</different></node>'
>>> # Get child values where name is 'same'
... tmp = arcom.get_child_values_by_name(n,'same')
>>> # Result is a list
... type(tmp)
<type 'list'>
>>> # Show result
... tmp
['firstEQ', 'secondEQ']
>>> # Get child values where name is 'different'
... tmp = arcom.get_child_values_by_name(n,'different')
>>> # Show result
... tmp
['DIFF']
```

- `datapoint_from_url(url_string, ssl_config = None)`
  Creates DataPoint from specified URL (`url_string`) with the specified SSL configuration (if `ssl_config` is present).

Example 17 Creating DataPoint from URL

```
>>> import arc
>>> import arcom
>>> tmpList = []
>>> status = ''
>>> # create DataPoint from a local directory
... dp = arcom.datapoint_from_url('file:///usr/local/share/arc')
>>> # list files
... (files, stat) = dp.ListFiles()
>>> # if it is not empty
... if files:
>>>     status = 'found'
>>>     # for all the entries, get type and name
...     for f in files:
>>>         if (f.GetType() == arc.FileInfo.file_type_file):
>>>             type = 'file'
>>>         elif (f.GetType() == arc.FileInfo.file_type_dir):
>>>             type = 'dir'
>>>         else:
>>>             type = 'unknown'
>>>         # get results together in a list
...         tmpList.append(f.GetName() + ' (' + type + ')\n')
>>> else:
>>>     status = 'Could not access data. Reason: %s' % str(stat)
>>>
>>> # see result
... str(stat)
'Operation completed successfully'
>>> # show list
... tmpList
['nordugrid.schema (unknown)\n']
```

- `parse_url(url)`
  Parses a URL. Gets protocol, host, port and path.

Example 18 Parsing a URL

```
>>> import arcom
>>> proto, host, port, path = arcom.parse_url('boo://no.one.here:123/foo')
>>> proto
'boo'
>>> host
'no.one.here'
>>> port
123
>>> path
'foo'
```

3.2 arcom.client

Module "arcom.client" contains a base Client class for sending SOAP messages to services.

- `Client(url, ns, print_xml = False, xmlnode_class = arc.XMLNode, ssl_config = {})`
  This is the constructor of the Client class.
  - url is the URL of the service, it could be a list of URLs

- ns contains the namespaces we want to use with each message
- `print_xml` is for debugging, prints all the SOAP messages to the screen
- `xmlnode_class` is the XML node class to be used; by default, it is "arc.XMLNode"
- `ssl_config` is the SSL configuration to be used for secure connection; it is a dictionary that contains information about client (`proxy_file` or `key_file` and `cert_file`) and CAs (`ca_file` or `ca_dir`)

- `call(tree, return_tree_only = False)`
  Creates a SOAP message from an XMLTree and sends it to the service.
  - tree is an XMLTree object containing the content of the request
  - `return_tree_only` indicates that we only need to put the response into an XMLTree

Example 19 Creating a client and calling the echo service (XMLTree)

```
>>> import arc
>>> import arcom
>>> # Import the Client class
... Client = arcom.import_class_from_string('arcom.client.Client')
>>> # Import the XMLTree class
... XMLTree = arcom.import_class_from_string('arcom.xmltree.XMLTree')
>>> # Create namespace - it will be used for the message sent
... ns = arc.NS({'echo':'urn:echo'})
>>> # Create client
... c = Client('http://your.server.example.com:50000/Echo',ns,print_xml=True)
>>> # Create message
... msg = XMLTree(from_tree = ('echo:echo',[('echo:say', 'Hello, World!')]))
>>> # Let the client do what it is meant for
... # Note that we created the client with print_xml=True
... # so both request and response will be displayed in an easy-to-read form
... c.call(msg)
Request:
        #   <soap-env:Body>
        #       <echo:echo>
        #           <echo:say>Hello, World!</echo:say>
        #       </echo:echo>
        #   </soap-env:Body>

Response:
        #   <soap-env:Envelope>
        #       <soap-env:Body>
        #           <echo:echoResponse>
        #               <echo:hear>[ Hello, World! ]</echo:hear>
        #           </echo:echoResponse>
        #       </soap-env:Body>
        #   </soap-env:Envelope>

'<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"><soap-en
v:Body><echo:echoResponse><echo:hear>[ Hello, World! ]</echo:hear></echo:echoResponse></s
oap-env:Body></soap-env:Envelope>'
```

- `call_raw(outpayload)`
  Send a POST request with the SOAP XML message.
  - outpayload is an XMLNode with the SOAP message

Example 20 Creating a client and calling the echo service (SOAP)

```
>>> import arc
>>> import arcom
>>> # Import the Client class
... Client = arcom.import_class_from_string('arcom.client.Client')
>>> # Create namespace - it will be used for the message sent
... ns = arc.NS({'echo':'urn:echo'})
>>> # Create client
... c = Client('http://arctest.ki.iif.hu:50000/Echo',ns,print_xml=True)
>>> # Create SOAP Payload
... pl = arc.PayloadSOAP(ns)
>>> # Create message structure and set content
... pl.NewChild('echo:echo',ns).NewChild('echo:say',ns).Set('Hello, World!')
>>> # Let client do the call; see response
... c.call_raw(pl)
'<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"><soap-en
v:Body><echo:echoResponse><echo:hear>[ Hello, World! ]</echo:hear></echo:echoResponse></s
oap-env:Body></soap-env:Envelope>'
```

3.3 arcom.logger

Module "arcom.logger" provides simple access to the logging capability of ARC.

- `log_levels`
  There are six log levels at the moment:
  - arc.DEBUG
  - arc.VERBOSE
  - arc.INFO
  - arc.WARNING
  - arc.ERROR
  - arc.FATAL

- `get_logger(system = '<UNKNOWN>')`
  Creates a logger - with root logger as a parent - on a given subdomain (system). If no such is given, it will be "`<UNKNOWN>`" by default.

Example 21 Using the Logger

```
>>> import arc
>>> import arcom
>>> import sys
>>> # Import get_logger function
... from arcom.logger import get_logger
>>> # Create logger
... rl = get_logger()
>>> rl.logger
<arc.Logger; proxy of <Swig Object of type 'Arc::Logger *' at 0x9ef3f60> >
>>> arclogger = rl.logger
>>> # Add new destination
... arclogger.addDestination(arc.LogStream(sys.stdout))
>>> # Log a message
... # Note that the message is also returned as an output
... rl.msg(arc.INFO, 'Hello, World!')
[2009-06-04 11:00:42] [Arc.<UNKNOWN>] [INFO] [13120/148748400] Hello, World!
'Hello, World!'
```

3.4 arcom.security

Module "arcom.security" provides tools for managing simple authorisation policies and requests. This includes policy format conversion (between storage and ARC policy), utility functions for accessing the decision making mechanism and retrieving credential information.

- `class AuthRequest`
  Represents simple authorisation request. Conversion is limited to ARCAuth format at the moment.
  - `__init__ - construction of AuthRequest`
    AuthRequest is created from the incoming message. Before the message reaches a service it is passed through several message chain components. (See ARC HED documentation[10] for details.) During this, information specific to the MCC is probably added to the message. Specifically, when it is passed through the TLS MCC, then security information is added. (See security documentation[11] for details about security design in ARC.) This information is then retrieved when AuthRequest is created.

Example 22 Identity part of subject retrieved when TLS is not used

```
<ra:Subject xmlns:ra="http://www.nordugrid.org/schemas/request-arc">
  ...
  <ra:SubjectAttribute
    AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/tls/identity"
    Type="string">ANONYMOUS</ra:SubjectAttribute>
</ra:Subject>
```

When TLS MCC is not included in the message chain, identity retrieval will fail, thus showing up with an anonymous user.

  - `get_request(self, action, format = 'ARCAuth')`
    Returns a request in a XML string for the given action. This action should be one of the storage actions. These are 'read', 'addEntry', 'removeEntry', 'delete', 'modifyPolicy', 'modifyStates' and 'modifyMetadata'. Format should be ARCAuth at the moment, as other formats are not supported yet.

  - `get_identity(self)`
    Returns identity information.

[10] `http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/doc/tech_doc/hed/ARCHED_article.pdf`
[11] `http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/doc/tech_doc/sec/SecurityFrameworkofARC1.tex`

- `get_identity_and_ca(self)`
  Returns identity and CA information.

- `class AuthPolicy`
  Class to set / retrieve / convert simple policies.

Example 23 Creating policy

```
>>> from arcom.security import AuthPolicy
>>> # create policy object
... p = AuthPolicy()
>>> # set policy for Alice and Bob
... # Alice is allowed to read and add entries (addEntry)
... # Bob is only allowed to read
... AlicePol = ('Alice','+read +addEntry')
>>> BobPol = ('Bob','+read')
>>> p.set_policy([AlicePol,BobPol])
>>> # see Policy document
... print p.get_policy('ARCAuth')
<Policy xmlns="http://www.nordugrid.org/schemas/policy-arc"
        CombiningAlg="Deny-Overrides">
  <Rule Effect="Permit">
    <Description>Alice is allowed to read, addEntry</Description>
    <Subjects>
      <Subject>
        <Attribute AttributeId="http://www.nordugrid.org/schemas/policy-arc/
types/tls/identity" Type="string">Alice</Attribute>
      </Subject>
    </Subjects>
    <Actions>
      <Action AttributeId="http://www.nordugrid.org/schemas/policy-arc/
types/storage/action" Type="string">read</Action>
      <Action AttributeId="http://www.nordugrid.org/schemas/policy-arc/
types/storage/action" Type="string">addEntry</Action>
    </Actions>
  </Rule>
  <Rule Effect="Permit">
    <Description>Bob is allowed to read</Description>
    <Subjects>
      <Subject>
        <Attribute AttributeId="http://www.nordugrid.org/schemas/policy-arc/
types/tls/identity" Type="string">Bob</Attribute>
      </Subject>
    </Subjects>
    <Actions>
      <Action AttributeId="http://www.nordugrid.org/schemas/policy-arc/
types/storage/action" Type="string">read</Action>
    </Actions>
  </Rule>
</Policy>
```

- `make_decision(policy, request)`
  Method for accessing decision making functionality of ARC. ARC Evaluator is used to make decision about the request according to supplied policy.

Example 24 Decision making

```
...
>>> dsa = DummySecAttr('Alice')
>>> # set 'thisown' to False to avoid problems
... dsa.thisown = False
>>>
>>> # export XML in ARCAuth format
... ex = dsa.Export()
>>>
>>> from arcom.security import AuthPolicy
>>> # create policy object
... p = AuthPolicy()
>>> # set policy for Alice
... # Alice is allowed to read and add entries (addEntry)
... AlicePol = ('Alice','+read +addEntry')
>>> p.set_policy([AlicePol])
>>>
>>> from arcom.security import make_decision
>>>
>>> # get policy XML
... px = p.get_policy()
>>> rx = ex.GetXML()
>>>
>>> decision = make_decision(px, rx)
>>>
>>> # Possible results:
... # arc.DECISION_PERMIT -- 0
... # arc.DECISION_DENY -- 1
... # arc.DECISION_INDETERMINATE -- 2
... # arc.DECISION_NOT_APPLICABLE -- 3
...
>>> decision
0
```

In this example, Alice requests read for a temporary file. This request is made via DummySecAttr, when Export is called. (Details about DummySecAttr can be found in Appendix B of this document.)

- `parse_ssl_config(cfg)`
  Method for processing an XML node and retrieve SSL configuration information. First it looks for a ClientSSLConfig child of the supplied node, then it either collects information from a file given in FromFile attribute or from other children of ClientSSLConfig, namely KeyPath, CertificatePath and CACertificatesDir.
  So if our ARC config looked like

```
<?xml version="1.0"?>
<ArcConfig
  xmlns="http://www.nordugrid.org/schemas/ArcConfig/2007"
  xmlns:tcp="http://www.nordugrid.org/schemas/ArcMCCTCP/2007"
>
    ...
    <Chain>
        ...
        <Service name="pythonservice" id="bartender">
            <ClassName>storage.bartender.bartender.BartenderService</ClassName>
            <LibrarianURL>https://localhost:60000/Librarian</LibrarianURL>
            <ClientSSLConfig FromFile="/etc/arc/clientsslconfig.xml" />
        </Service>
        ...
```

```
    </Chain>
</ArcConfig>
```

and "/etc/arc/clientsslconfig.xml" contained the following lines:

```
<?xml version="1.0"?>
<ClientSSLConfig>
    <KeyPath>/etc/grid-security/hostkey.pem</KeyPath>
    <CertificatePath>/etc/grid-security/hostcert.pem</CertificatePath>
    <CACertificatesDir>/etc/grid-security/certificates</CACertificatesDir>
</ClientSSLConfig>
```

then parsing would produce results like in the example below.

Example 25 SSL config example

```
>>> import arc
>>> import arcom
>>> from arcom.security import parse_ssl_config
>>>
>>> xml_str = file('/etc/arc/ssl_config_example.xml').read()
>>> configNode = arc.XMLNode(xml_str)
>>>
>>> serviceNode = configNode.Get('Chain').Get('Service')
>>>
>>> print parse_ssl_config(serviceNode)
{'key_file': '/etc/grid-security/hostkey.pem',
 'cert_file': '/etc/grid-security/hostcert.pem',
 'ca_file': '/etc/grid-security/certificates'}
```

3.5 arcom.service

Module "arcom.service" provides the Service class and other tools for service development. The Service class deals with Trust Manager and SSL configuration and has the process method to get requests from incoming messages and create outgoing ones from the results. This class should be extended when creating a new service. Note that security related parts will not be discussed here; for those please see the security documentation[12] of ARC.

- `Service`
  Service class will be presented through an example, in which DummyService will provide access to the funcionality of the Dummy class. (Complete source code can be found in Appendix C of this document.)
  Note that a small change in AuthRequest class (found in "security.py") is required for this example to work because the incoming message in our example could not be processed in a way like on the server side. Therefore it is assumed that the `__init__` method of AuthRequest has:

```
auth = message.Auth()
import arc
try:
    xml = auth.Export(arc.SecAttr.ARCAuth)
    subject = xml.Get('RequestItem').Get('Subject')
except:
    subject = arc.XMLNode(arc.NS({'ra':request_ns}), 'ra:Subject')
```

  instead of

[12] http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/doc/tech_doc/sec/SecurityFrameworkofARC1.tex

```
auth = message.Auth()
import arc
xml = auth.Export(arc.SecAttr.ARCAuth)
subject = xml.Get('RequestItem').Get('Subject')
```

so an empty Subject is available if auth is not present.

Example 26 Using DummyService - an example service based on arcom.service.Service

```
>>> # Reminder:
... # import arc
... #
... # class Dummy:
... #   def foo(self, foomsg = ''):
... #     return ''.join(['Foo message is: ',str(foomsg)])
... #
... # from arcom.service import Service
... #
... # class DummyService(Service):
... #   """ DummyService class based on Service class in arcom.service """
... #   ...
...
>>> # Create instance of DummyService
... ds = DummyService(None)
>>> # Create namespace - it will be used for the message sent
... ns = arc.NS({'dmmy':'urn:foo'})
>>> # Create SOAP Envelope
... se = arc.SOAPEnvelope(ns)
>>> # Create content
... se.NewChild('dmmy:foo').Set('Foo!')
(None, 'Foo!')
>>> # Create SOAP Message
... msg = arc.SOAPMessage()
>>> # Payload
... pls = arc.PayloadSOAP(se)
>>> # Set Message Payload
... msg.Payload(pls)
>>> # create outgoing message that will hold response
... outse = arc.SOAPEnvelope(ns)
>>> outmsg = arc.SOAPMessage()
>>> outpls = arc.PayloadSOAP(outse)
>>> outmsg.Payload(outpls)
>>> # call service
... ds.process(msg, outmsg)
<arc.MCC_Status; proxy of <Swig Object of type 'Arc::MCC_Status *' at 0x9cd8228> >
>>> # show result
... print outmsg.Payload().GetXML()
<soap-env:Envelope xmlns:dmmy="urn:foo" xmlns:soap-enc="http://schemas.xmlsoap.o
rg/soap/encoding/" xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/" xm
lns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XML
Schema-instance"><soap-env:Body><dmmy:fooResponse>Foo message is: Foo!</dmmy:foo
Response></soap-env:Body></soap-env:Envelope>
```

- `ServiceState`
  ServiceState is part of the basic LIDI[13] for Python services. This basic implementation provides information about whether a service is up and running. XML representation of state could be retrieved by calling the "GetLocalInformation" method found in the Service class. This will be

[13] Local Information Description Interface; see: `http://www.knowarc.eu/documents/Knowarc_D1.2-1_07.pdf`

presented through an example, in which DummyService will provide information regarding its state. (Complete source code can be found in Appendix C of this document.)
Note that a change in AuthRequest class (found in "security.py") is required for this example to work (see 3.5).

Example 27 Retrieving state of DummyService

```
>>> # Reminder:
... # import arc
... #
... # ...
... #
... # from arcom.service import Service
... #
... # class DummyService(Service):
... #   """ DummyService class based on Service class in arcom.service """
... #
... #   ...
... #
... #   def status(self, inpayload):
... #       # get local info
... #       fooinfo = self.GetLocalInformation()
... #
... #       # get service status
... #       foostatus = str(fooinfo.Get('AdminDomain').Get('Services').Get('Service').\
... #       Get('Endpoint').Get('ServingState'))
... #       ...
...
>>> # Create instance of DummyService
... ds = DummyService(None)
>>> # Create namespace - it will be used for the message sent
... ns = arc.NS({'dmmy':'urn:foo'})
>>> # Create SOAP Envelope
... se = arc.SOAPEnvelope(ns)
>>> # Create content
... se.NewChild('dmmy:status')
<arc.XMLNode; proxy of <Swig Object of type 'Arc::XMLNode *' at 0x9cb8af8> >
>>> # Create SOAP Message
... msg = arc.SOAPMessage()
>>> # Payload
... pls = arc.PayloadSOAP(se)
>>> # Set Message Payload
... msg.Payload(pls)
>>> # create outgoing message that will hold response
... outse = arc.SOAPEnvelope(ns)
>>> outmsg = arc.SOAPMessage()
>>> outpls = arc.PayloadSOAP(outse)
>>> outmsg.Payload(outpls)
>>> # call service
... ds.process(msg, outmsg)
<arc.MCC_Status; proxy of <Swig Object of type 'Arc::MCC_Status *' at 0x9cdb090> >
>>> # show result
... print outmsg.Payload().GetXML()
<soap-env:Envelope xmlns:dmmy="urn:foo" xmlns:soap-enc="http://schemas.xmlsoap.o
rg/soap/encoding/" xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/" xm
lns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XML
Schema-instance"><soap-env:Body><dmmy:statusResponse>Status: production</dmmy:st
atusResponse></soap-env:Body></soap-env:Envelope>
```

- `parse_node(node, names, single = False, string = True)`
  Calls `node_to_data()` for each child of the given node.
  - `node` is the XMLNode whose children we want to convert
  - `names` is a list of tag names which will be returned in the specified order
  - `single` indicates whether only one value is needed beside the key; if so, do not put it into a list
  - `string` indicates whether string contents of nodes are needed - not the nodes themselves

Example 28 parse_node

```
>>> from arc import XMLNode
>>> from arcom.service import parse_node
>>>
>>> # create xml node
... xml = XMLNode('''
... <myList>
...   <myElement>
...     <myID>0</myID>
...     <LN>/</LN>
...   </myElement>
...   <myElement>
...     <myID>1</myID>
...     <LN>/testfile</LN>
...   </myElement>
... </myList>''')
>>>
>>> # default: single = False and string = True
... parse_node(xml, ['myID','LN'])
{'1': ['/testfile'], '0': ['/']}
>>> # single = True
... parse_node(xml, ['myID','LN'], single = True)
{'1': '/testfile', '0': '/'}
>>> parse_node(xml, ['myID','LN'], True)
{'1': '/testfile', '0': '/'}
>>> # string = False
... parse_node(xml, ['myID','LN'], string = False)
{<arc.XMLNode; proxy of <Swig Object of type 'Arc::XMLNode *' at 0x996f040> >: [
<arc.XMLNode; proxy of <Swig Object of type 'Arc::XMLNode *' at 0x996f050> >], <
arc.XMLNode; proxy of <Swig Object of type 'Arc::XMLNode *' at 0x9997590> >: [<a
rc.XMLNode; proxy of <Swig Object of type 'Arc::XMLNode *' at 0x99975a0> >]}
```

- `parse_to_dict(node, names)`
  Converts the children of the node to a dictionary of dictionaries.
  - `node` is the XMLNode whose children we want to convert
  - `names` is a list of tag names; for each child only these names will be included in the dictionary

  Note that the first element of "names" is considered as sthe child whose value will serve as the key of dictionary. Other elements are put in an inner dictionary with their names being the key and their content being the value for that key.

Example 29 parse_to_dict

```
>>> from arc import XMLNode
>>> from arcom.service import parse_to_dict
>>>
>>> # create xml node
... xml = XMLNode('''
... <myList>
...   <myElement>
...     <myID>123</myID>
...     <refID>abc</refID>
...     <state>alive</state>
...     <size>123456</size>
...   </myElement>
...   <myElement>
...     <myID>456</myID>
...     <refID>fed</refID>
...     <state>alive</state>
...     <size>987</size>
...   </myElement>
... </myList>''')
>>>
>>> # example 1 - 'myID' as key
... # 'state' and 'size' goes to the inner dictionary
... parse_to_dict(xml, ['myID','state','size'])
{'123': {'state': 'alive', 'size': '123456'}, '456': {'state': 'alive', 'size':
'987'}}
>>>
>>> # example 2 - 'myID' as key
... # 'myID' also goes to the inner dictionary
... parse_to_dict(xml, ['myID','myID','state','size'])
{'123': {'myID': '123', 'state': 'alive', 'size': '123456'}, '456': {'myID': '45
6', 'state': 'alive', 'size': '987'}}
```

- `create_response(method_name, tag_names, elements, payload, single = False)`
  Creates a SOAP XML payload from a dictionary of tag names and list of values.
  - `method_name` is the name of the method which will be used as a prefix in the name of the "Response" tag
  - `tag_names` is a list of names which will be used in the specified order as tag names
  - `elements` is a dictionary where the key will be tagged as the first tag name and the value is a list whose items will be tagged in the order of the `tag_names` list
  - `payload` is an XMLNode that the response will be added to
  - `single` indicates whether there is only one value per key

Example 30 create_response

```
>>> import arc
>>> from arcom.service import create_response
>>>
>>> method_name = 'dummy'
>>> # create tag_names; 'myID' will be the key
... tag_names = ['myID', 'state', 'size']
>>> # create elements
... # elements of the lists in this dictionary are values and will be tagged
... # according to names found in tag_names except for the first element
... # e.g.: 'alive' will be tagged with 'state'
... elements = {'123': ['alive', '123456'], '456': ['alive', '987']}
>>> # create empty payload
... payload = arc.PayloadSOAP(arc.NS())
>>>
>>> # create response
... response = create_response(method_name, tag_names, elements, payload)
>>>
>>> # show it
... print response.GetXML(True)
<soap-env:Envelope xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding/" xm
lns:soap-env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w
3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap-env:Body>
    <dummyResponse>
      <dummyResponseList>
        <dummyResponseElement>
          <myID>123</myID>
          <state>alive</state>
          <size>123456</size>
        </dummyResponseElement>
        <dummyResponseElement>
          <myID>456</myID>
          <state>alive</state>
          <size>987</size>
        </dummyResponseElement>
      </dummyResponseList>
    </dummyResponse>
  </soap-env:Body>
</soap-env:Envelope>
```

- `node_to_data(node, names, single = False, string = True)`
  Get some children of an XMLNode and return them in a list in the specified order using the first one as a key.
  - `node` is an XMLNode that has some children
  - `names` is a list of the names of those children we want to extract; first string in this list will always be the key
  - `single` indicates whether only one value is needed; if so, do not put it into a list
  - `string` indicates whether string contents of nodes are needed - not the nodes themselves

Example 31 node_to_data - 1

```
>>> from arc import XMLNode
>>> from arcom.service import node_to_data
>>>
>>> # create node
... xml = XMLNode('''
... <myElement>
...   <myID>0</myID>
...   <refID>abc</refID>
...   <section>states</section>
...   <property>spares</property>
...   <value>2</value>
...   <dummy>foo</dummy>
...   <changeType>set</changeType>
... </myElement>
... ''')
>>>
>>> # create names
... # 'myID' will be the key
... # all the other names will be in a list in the specified order
... # 'dummy' will be left out
... names = ['myID', 'refID', 'changeType', 'section', 'property', 'value']
>>>
>>> # call node_to_data
... node_to_data(xml,names)
('0', ['abc', 'set', 'states', 'spares', '2'])
```

Example 32 node_to_data - 2

```
>>> from arc import XMLNode
>>> from arcom.service import node_to_data
>>>
>>> # create node
... xml = XMLNode('''
... <getRequest>
...   <GUID>11</GUID>
...   <myID>99</myID>
... </getRequest>
... ''')
>>>
>>> # create names
... # 'myID' will be the key
... names = ['myID', 'GUID']
>>>
>>> # call node_to_data
... # GUID is the single value so do not put it in a list
... node_to_data(xml,names, True)
('99', '11')
```

- `get_data_node(node)`
  Gets the data node from the `Body` of a SOAP message where the first child node of the `Body` refers to the method being called. E.g.:

  ```
  Request:
          <soap-env:Envelope>
            <soap-env:Body>
  ```

```
            <method>
              <data>Value</data>
            </method>
          </soap-env:Body>
        </soap-env:Envelope>

Data node:
          <data>Value</data>
```

Example 33 get_data_node

```
>>> from arc import XMLNode
>>> from arcom.service import get_data_node
>>>
>>> # create node
... xml = XMLNode('''
... <Envelope>
...   <Body>
...     <method>
...       <data>Value</data>
...     </method>
...   </Body>
... </Envelope>
... ''')
>>>
>>> # call get_data_node
... dn = get_data_node(xml)
>>>
>>> # show result
... dn.GetXML()
'<data>Value</data>'
```

3.6 arcom.threadpool

Module "arcom.threadpool" provides simple thread pool support. Consists of classes ThreadPool, ThreadPoolThread and ReadWriteLock. ThreadPool could be used to create a pool of threads, then it accepts tasks that will be dispatched to the next available thread. ThreadPoolThread could be used to retrieve a task and execute it. ReadWriteLock is a lock object that allows many simultaneous "read locks", but only one "write lock."

- `class ThreadPool`
  ThreadPool is a flexible thread pool class. It creates a pool of threads, then accepts tasks that will be dispatched to the next available thread.
  - `setThreadCount(self, newNumThreads)`
    Sets the current pool size. It does so by acquiring the resizing lock, then calling the private version of this method to grow or shrink the pool.
  - `getThreadCount(self)`
    Returns the number of threads in the pool.
  - `queueTask(self, task, args=None, taskCallback=None)`
    Inserts a task into the queue. Task must be callable; args and taskCallback may be None.
  - `getNextTask(self)`
    Retrieves the next task from the task queue. For use only by ThreadPoolThread objects contained in the pool.

  - `joinAll(self, waitForTasks = True, waitForThreads = True)`
    Clears the task queue and terminates all pooled threads, optionally allowing the tasks and threads to finish.

- `class ThreadPoolThread(threading.Thread)`
  ThreadPoolThread is a pooled thread class.

  - `run(self)`
    Until told to quit, retrieve the next task and execute it, calling the callback if any.

  - `goAway(self)`
    Exit the run loop next time through.

- `class ReadWriteLock`
  ReadWriteLock is a lock object that allows many simultaneous "read locks", but only one "write lock".

  - `acquire_read(self)`
    Acquires a read lock. Blocks only if a thread has acquired the write lock.

  - `release_read(self)`
    Releases a read lock.

  - `acquire_write(self)`
    Acquires a write lock. Blocks until there are no acquired read or write locks.

  - `release_write(self)`
    Releases a write lock.

Two examples are pressented below. In the first one, four waitTasks with different wait times are queued while there are three threads. In the second one, a counter is incremented by four addTasks on four threads. (Here a ReadWriteLock is used to make sure that only one thread has write access to the counter at a time.) (Complete source code can be found in Appendix D of this document.)

Example 34 arcom.threadpool

```
$./threadpooltest.py

Example1 - waitTask
(1): WaitTask starting
WaitTask sleeping for 4 seconds
(2): WaitTask starting
WaitTask sleeping for 1 seconds
(3): WaitTask starting
WaitTask sleeping for 6 seconds
Callback called for Waiter (2)
(4): WaitTask starting
WaitTask sleeping for 2 seconds
Callback called for Waiter (4)
Callback called for Waiter (1)
Callback called for Waiter (3)

Example2 - addTask
COUNTER before join:  0
(5): Added 10 to counter, counter is now 10
(6): Added 10 to counter, counter is now 20
(7): Added 10 to counter, counter is now 30
(8): Added 10 to counter, counter is now 40
(6): Added 10 to counter, counter is now 50
(5): Added 10 to counter, counter is now 60
(7): Added 10 to counter, counter is now 70
(8): Added 10 to counter, counter is now 80
(5): Added 10 to counter, counter is now 90
(7): Added 10 to counter, counter is now 100
(6): Added 10 to counter, counter is now 110
(8): Added 10 to counter, counter is now 120
Callback called for ('addTask', 10)
Callback called for ('addTask', 10)
Callback called for ('addTask', 10)
Callback called for ('addTask', 10)
COUNTER after join:  120
```

3.7 arcom.XMLTree

Module "arcom.XMLTree" provides the XMLTree class to convert from XML to native Python structures and vice versa. Furthermore, it provides utility functions and some basic query methods for XMLTree structures.

- `class XMLTree`
  With the help of the XMLTree class XML can be converted to native Python structures and vice versa. It also provides some useful functions to handle these structures.
  - `XMLTree(from_node = None, from_string = '', from_tree = None, rewrite = {}, forget_namespace = False)`
    - `from_node` is an XMLNode that could be converted to XMLTree
    - `from_string` is a string representation of an XMLNode that could be converted to XMLTree
    - `from_tree` is a tree structure or an XMLTree object that could be converted to XMLTree
    - `rewrite` is a dictionary; if an XML node has a name which is a key in this dictionary then it will be renamed as the value of that key; note though, when `from_tree` is used, this parameter is ignored

* `forget_namespace` tells whether the XMLTree should not contain the namespace prefixes; note though, when `from_tree` is used, this parameter is ignored

The parameter `from_tree` has the highest priority; if it is not None, then `from_string` and `from_node` are ignored. If `from_tree` is None but `from_node` is given, then `from_string` is ignored. If only `from_string` is given, then it will be the choosen one.

Some examples of creating an XMLTree are shown below.

Example 35 Creating an XMLTree

```
>>> from arc import XMLNode
>>> from arcom.xmltree import XMLTree
>>>
>>> # create XMLTree sources
... # xmlstr - will be used to create an XMLTree from string
... # xmlnodestr - will be used to create 'node' (an XMLNode)
... # node - will be used to create an XMLTree from XMLNode
... # treestruct - for creating XMLTree from tree structure
... xmlstr = '<echo><say>Hello!</say></echo>'
>>> xmlnodestr = '<node><subnode>Subnode</subnode></node>'
>>> node = XMLNode(xmlnodestr)
>>> treestruct = ('root', [('leaf','Leaf')])
>>>
>>> # from_tree has the highest priority
... t1 = XMLTree(from_node = node, from_string = xmlstr, from_tree = treestruct)
>>> t1.get()
[('root', [('leaf', 'Leaf')])]
>>>
>>> # no from_tree supplied; from_node will be used
... t2 = XMLTree(from_node = node, from_string = xmlstr)
>>> t2.get()
[('node', [('subnode', 'Subnode')])]
>>>
>>> # from_string only
... t3 = XMLTree(from_string = xmlstr)
>>> t3.get()
[('echo', [('say', 'Hello!')])]
```

Example 36 XMLTree - forget namespace

```
>>> from arc import XMLNode
>>> from arcom.xmltree import XMLTree
>>>
>>> # Create XMLTree objects to demonstrate
... # the use of forget namespace feature
... # node - is an XMLNode
... # tree - is an XMLTree that is created from 'node'
... node = XMLNode('<soap-env:Envelope xmlns:hash="urn:hash" \
... 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">\
... <soap-env:Body>\
... <hash:get>\
... <hash:IDs>\
... <hash:ID>0</hash:ID>\
... <hash:ID>1</hash:ID>\
... <hash:ID>2</hash:ID>\
... </hash:IDs>\
... </hash:get>\
... </soap-env:Body>\
... </soap-env:Envelope>')
>>>
>>> tree = XMLTree(node)
>>> # show tree structure
... print tree
('soap-env:Envelope', [('soap-env:Body', [('hash:get', [('hash:IDs', [('hash:ID
', '0'), ('hash:ID', '1'), ('hash:ID', '2')])])])])
>>>
>>> # forget_namespace is ignored when from_tree is used
... tree2 = XMLTree(from_tree = tree, forget_namespace = True)
... print tree2
('soap-env:Envelope', [('soap-env:Body', [('hash:get', [('hash:IDs', [('hash:ID
', '0'), ('hash:ID', '1'), ('hash:ID', '2')])])])])
>>>
>>> # forget_namespace with from_node
... tree3 = XMLTree(from_node = node, forget_namespace = True)
>>> # it works as expected
... print tree3
('Envelope', [('Body', [('get', [('IDs', [('ID', '0'), ('ID', '1'), ('ID', '2')
])])])])
>>>
>>> # forget_namespace with from_string
... strnode = '<myns:foo xmlns:myns="urn:foo"><myns:dummy>Hello, World!</myns:dummy>\
... </myns:foo>'
>>> tree4 = XMLTree(from_string = strnode, forget_namespace = True)
>>> # it also works
... print tree4
('foo', [('dummy', 'Hello, World!')])
```

Some examples of creating an XMLTree with the rewrite feature are shown below.

Example 37 XMLTree - rewrite

```
>>> from arc import XMLNode
>>> from arcom.xmltree import XMLTree
>>>
>>> # Create XMLTree objects to demonstrate
... # the use of rewrite feature
>>> strnode = '<myns:foo xmlns:myns="urn:foo"><myns:dummy>Hello, World!</myns:dummy>\
... </myns:foo>'
>>> tree = XMLTree(from_string = strnode,\
...                 rewrite = {'myns:foo':'myns:boo'},\
...                 forget_namespace = False)
>>>
>>> # 'myns:foo' is turned into 'myns:boo'
... print tree
('myns:boo', [('myns:dummy', 'Hello, World!')])
>>>
>>> # combine with forget_namespace
... tree2 = XMLTree(from_string = strnode,\
...                  rewrite = {'foo':'boo'},\
...                  forget_namespace = True)
>>> # 'forget_namespace' is True so namespace prefix 'myns' is being stripped off
... # ('myns:foo' -> 'foo'; 'myns:dummy' -> 'dummy')
... # then 'foo' is turned into 'boo'
... print tree2
('boo', [('dummy', 'Hello, World!')])
```

- `add_to_node(self, node, path = None)`
  Adds a tree structure to an XMLNode. Structure is added as a child of the target XMLNode.
  - `node` is the target XMLNode
  - `path` selects the part of the XMLTree to be added

Example 38 XMLTree - add_to_node

```
>>> from arc import XMLNode
>>> from arcom.xmltree import XMLTree
>>>
>>> node = XMLNode('<node/>')
>>>
>>> # create XMLTree
... tree = XMLTree(from_string = '<root><dummy/><hello>World</hello></root>')
>>>
>>> # from 'tree' add 'dummy' to 'node'
... tree.add_to_node(node, '/root/dummy')
>>>
>>> # show result
... print node.GetXML(True)
<node>
  <dummy></dummy>
</node>
>>>
>>> # from 'tree' add 'hello' to 'dummy' in 'node'
... dummy = node.Get('dummy')
>>> tree.add_to_node(dummy, '/root/hello')
>>>
>>> # show result
... print node.GetXML(True)
<node>
  <dummy><hello>World</hello></dummy>
</node>
```

- `pretty_xml(self, indent = ' ', path = None, prefix = '')`
  Returns a nicely formatted XML representation of the structure.
  - `indent` is a string that child nodes are indented with
  - `path` selects the part of the XMLTree to be shown. Full structure is shown when `path` is None.
  - `prefix` is a string that is put to the beginning of each line

Example 39 XMLTree - pretty_xml

```
>>> from arcom.xmltree import XMLTree
>>>
>>> # create XMLTree
... tree = XMLTree(from_string = '<root><Hello>World</Hello></root>')
>>>
>>> # select entire tree
... # indent children with space character
... # apply prefix for every new line
... print tree.pretty_xml(indent=' ', path='', prefix='# ')
# <root>
#  <Hello>World</Hello>
# </root>
>>>
>>> # note that it is possible to select multiple parts
... # create an example XMLTree to demonstrate that
... tree2 = XMLTree(from_string = '<root><sub><Hello>World</Hello></sub>\
... <sub><dummy>foo</dummy></sub></root>')
>>>
>>> # show '/root/sub'
... print tree2.pretty_xml(indent=' ', path='/root/sub', prefix='# ')
# <sub>
#  <Hello>World</Hello>
# </sub>
# <sub>
#  <dummy>foo</dummy>
# </sub>
```

- `__str__(self)`
  Returns string representation of the structure. Each node is represented by a dictionary where the name of the node serves as the key and the value for this key is a list of child nodes of that node.

Example 40 XMLTree - __str__

```
>>> from arcom.xmltree import XMLTree
>>>
>>> # create XMLTree
... tree = XMLTree(from_string = '<root><dummy>foo</dummy></root>')
>>>
>>> # show tree
... # print will result in __str__ being called
... # when trying to convert XMLTree to string
... print tree
('root', [('dummy', 'foo')])
```

- `get(self, path = None)`
  Returns the parts of the XMLTree that match `path`. If `path` is not given, it defaults to the root node. This function always returns a list.

Example 41 XMLTree - get

```
>>> from arcom.xmltree import XMLTree
>>>
>>> # create tree structure
... tree = \
... ('root',\
...   [('trunk',\
...     [('leaf','1'),\
...      ('leaf','2')]\
...   )]\
... )
>>>
>>> # create XMLTree from tree structure
... xt = XMLTree(from_tree = tree)
>>>
>>> # if called with path being None
... # get() selects the root node
... xt.get()
[('root', [('trunk', [('leaf', '1'), ('leaf', '2')])])]
>>>
>>> # Path is just a plain path.
... # Empty tag name matches everything, so - in this particular case -
... # all expressions below will produce the same result.
... xt.get('/root/trunk/leaf')
[('leaf', '1'), ('leaf', '2')]
>>> xt.get('//trunk/leaf')
[('leaf', '1'), ('leaf', '2')]
>>> xt.get('/root//leaf')
[('leaf', '1'), ('leaf', '2')]
>>> xt.get('/root/trunk/')
[('leaf', '1'), ('leaf', '2')]
>>> xt.get('///leaf')
[('leaf', '1'), ('leaf', '2')]
>>> xt.get('///')
[('leaf', '1'), ('leaf', '2')]
```

– `get_trees(self, path = None)`
Returns XMLTree object for each subtree that match `path`. This function always returns a list.

Example 42 XMLTree - get_trees

```
>>> from arcom.xmltree import XMLTree
>>>
>>> # create tree structure
... tree = \
... ('trunk',\
...   [( 'branch',\
...      [('leaf','1'),\
...       ('leaf','2')]\
...    ),\
...    ( 'branch',\
...      [('leaf','3')]\
...    )]\
... )
>>>
>>> # create XMLTree from tree structure
... xt = XMLTree(from_tree = tree)
>>>
>>> # get branches ('/trunk/branch') in a list
... blist = xt.get_trees('/trunk/branch')
>>>
>>> # show branches in blist
... for b in blist:
...     print b
...
('branch', [('leaf', '1'), ('leaf', '2')])
('branch', [('leaf', '3')])
>>> # get leaves ('/trunk/branch/leaf') in a list
... llist = xt.get_trees('/trunk/branch/leaf')
>>>
>>> # show leaves in llist
... for l in llist:
...     print l
...
('leaf', '1')
('leaf', '2')
('leaf', '3')
```

- `get_value(self, path = None, *args)`
  Returns the value of the selected part. This means that if node N is the first node matched by `path` then the value of N (i.e. the list of its child nodes or its content if it has no child) is returned. If there is no match, and a default value is given, it will be returned instead.

Example 43 XMLTree - get_value

```
>>> from arcom.xmltree import XMLTree
>>>
>>> # create tree structure
... ts = ('branch',[('leaf','1')])
>>>
>>> # create XMLTree from tree structure
... tree = XMLTree(from_tree = ts)
>>>
>>> # get_value for '/branch/leaf'
... # will return the value from 'leaf'
... tree.get_value('/branch/leaf')
'1'
>>>
>>> # get_value for '/branch'
... # will return the value from 'branch'
... # ie the list of its children
... tree.get_value('/branch')
[('leaf', '1')]
>>>
>>> # get_value for a path that does not match
... # provide a default value of 'N/A'
... tree.get_value('/dummy','N/A')
'N/A'
```

- `add_tree(self, tree, path = None)`
  Adds a new subtree to a path. This will actually add `tree` to the first node that matches `path`. (Root node of `tree` will be a new child of the node matched by `path`.)

Example 44 XMLTree - add_tree

```
>>> from arcom.xmltree import XMLTree
>>>
>>> # create tree structure
... ts = \
... ('root',\
...   [ ('branch',\
...       [ ('leaf','1'),\
...         ('leaf','2')\
...       ]\
...     ),\
...     ('branch',\
...       [ ('leaf','3'),\
...         ('leaf','4')\
...       ]\
...     )\
...   ]\
... )
>>>
>>> # create XMLTree from tree structure
... tree = XMLTree(from_tree = ts)
>>>
>>> # add a new leaf ('leaf','5') to '/root/branch'
... # the new leaf will be added to the first node
... # matched by path ('/root/branch')
... tree.add_tree(('leaf','5'), '/root/branch')
>>>
>>> # show tree
... print tree
('root', [('branch', [('leaf', '1'), ('leaf', '2'), ('leaf', '5')]), ('branch',
[('leaf', '3'), ('leaf', '4')])])
```

- `get_values(self, path = None)`
  Returns the value (i.e. the list of child nodes or the content if one has no child) for all matched nodes selected by `path`. This function always returns a list. Note that unlike `get_value` this one takes all matched nodes into account. Furthermore this one does not accept a default value to return.

Example 45 XMLTree - get_values

```
>>> from arcom.xmltree import XMLTree
>>>
>>> # create tree structure
... ts = \
... ('root',\
...   [ ('branch',\
...       [ ('leaf','1'),\
...         ('leaf','2')\
...       ]\
...     ),\
...     ('branch',\
...       [ ('leaf','3'),\
...         ('leaf','4')\
...       ]\
...     )\
...   ]\
... )
>>>
>>> # create XMLTree from tree structure
... tree = XMLTree(from_tree = ts)
>>>
>>> # get values of branches ('/root/branch') in a list
... # this will return the lists of children of the branches
... tree.get_values('/root/branch')
[[('leaf', '1'), ('leaf', '2')], [('leaf', '3'), ('leaf', '4')]]
>>>
>>> # get values of leaves ('/root/branch/leaf') in a list
... # this will return the values of leaves
... tree.get_values('/root/branch/leaf')
['1', '2', '3', '4']
>>>
>>> # try a path that does not match
... # this will return an empty list
... tree.get_values('/dummy')
[]
```

- `get_dict(self, path = None, keys = {})`
  This method is designed to restore a set of `(key, value)` pairs stored in an XML structure. It returns a dictionary from the first node matched by `path`. `keys` is a dictionary which filters and translates the keys (e.g. if `keys` is 'my:node':'node', it will only return 'my:node', and will call it 'node'). The value for this key will be the text content of that node or the list of its child nodes. (Note though that `get_dict` was designed with simple text content in mind.)

Example 46 XMLTree - get_dict

```
>>> from arcom.xmltree import XMLTree
>>>
>>> # create tree structure
... ts =\
... ('person',\
...   [ ('name', 'Alice'),\
...     ('id', '11'),\
...     ('job', 'Librarian')\
...   ]
... )
>>>
>>> # create XMLTree from tree structure
... tree = XMLTree(from_tree = ts)
>>>
>>> # get dictionary from '/person'
... # only name and job is to be returned
... tree.get_dict('/person',{'name':'name','job':'job'})
{'job': 'Librarian', 'name': 'Alice'}
```

- `get_dicts(self, path = None, keys = {})`
  This method is designed to restore multiple sets of `(key, value)` pairs stored in an XML structure. It returns a list of dictionaries from nodes matched by `path`. `keys` is a dictionary which filters and translates the keys (e.g. if `keys` is 'my:node':'node', it will only return 'my:node', and will call it 'node'). The value for this key will be the text content of that node or the list of its child nodes. (Note though that `get_dict` was designed with simple text content in mind.)

Example 47 XMLTree - get_dicts

```
>>> from arcom.xmltree import XMLTree
>>>
>>> # create tree structure
... ts =\
... ('persons',\
...   [ ('person',\
...       [ ('name', 'Alice'),\
...         ('id', '11'),\
...         ('job', 'Librarian')\
...       ]\
...     ),\
...     ('person',\
...       [ ('name', 'Bob'),\
...         ('id', '12'),\
...         ('job', 'Bartender')\
...       ]\
...     )\
...   ]\
... )
>>>
>>> # create XMLTree from tree structure
... tree = XMLTree(from_tree = ts)
>>>
>>> # get dictionaries for '/persons/person'
... tree.get_dicts('/persons/person')
[{'job': 'Librarian', 'name': 'Alice', 'id': '11'},
{'job': 'Bartender', 'name': 'Bob', 'id': '12'}]
```

Appendices

A Helper classes for LogStream function

```
%inline %{
class CPyOutbuf : public std::streambuf
{
public:
     CPyOutbuf(PyObject* obj) {
         m_PyObj = obj;
         Py_INCREF(m_PyObj);
     }
     ~CPyOutbuf() {
         Py_DECREF(m_PyObj);
     }
protected:
     int_type overflow(int_type c) {
         // Call to PyGILState_Ensure ensures there is Python
         // thread state created/assigned.
         PyGILState_STATE gstate = PyGILState_Ensure();
         PyObject_CallMethod(m_PyObj, (char*) "write", (char*) "c", c);
         PyGILState_Release(gstate);
         return c;
     }
     std::streamsize xsputn(const char* s, std::streamsize count) {
         // Call to PyGILState_Ensure ensures there is Python
         // thread state created/assigned.
         PyGILState_STATE gstate = PyGILState_Ensure();
         PyObject_CallMethod(m_PyObj, (char*) "write", (char*) "s#", s, int(count));
         PyGILState_Release(gstate);
         return count;
     }
     PyObject* m_PyObj;
};

class CPyOstream : public std::ostream
{
public:
     CPyOstream(PyObject* obj) : m_Buf(obj), std::ostream(&m_Buf) {}
private:
     CPyOutbuf m_Buf;
};

%}
```

B Dummy SecAttr made for use with make_decision

```
import arc
from arc import SecAttr
import sys

class DummySecAttr (SecAttr):
  "Minimalistic implementation of SecAttr."
  def __init__(self, identity = 'Anonymous'):
    SecAttr.__init__(self)
    self.identity = identity
```

```
  def Export(self, format = SecAttr.ARCAuth):
    if format != SecAttr.ARCAuth:
      return None
    else:
      ns = arc.NS({'ra':'http://www.nordugrid.org/schemas/request-arc'})
      res = arc.XMLNode(ns, 'ra:Request')
      ritem = res.NewChild('ra:RequestItem')
      sub = ritem.NewChild('ra:Subject')
      subattr = sub.NewChild('ra:SubjectAttribute')
      subattr.Set(self.identity)
      subattr.NewAttribute('Type').Set('string')
      subattr.NewAttribute('AttributeId').Set(\
      'http://www.nordugrid.org/schemas/policy-arc/types/tls/identity'\
      )
      # request read for temporary file
      rsrc = ritem.NewChild('ra:Resource')
      rsrc.Set('file:///home/test')
      rsrc.NewAttribute('Type').Set('string')
      rsrc.NewAttribute('AttributeId').Set('urn:arc:resource:file')
      ac = ritem.NewChild('ra:Action')
      ac.Set('read')
      ac.NewAttribute('Type').Set('string')
      ac.NewAttribute('AttributeId').Set(\
      'http://www.nordugrid.org/schemas/policy-arc/types/storage/action'\
      )
      return res
```

C Dummy and DummyService - an example service based on arcom.service.Service

```
import arc

class Dummy:
  """ Dummy class that provides functionality """
  def foo(self, foomsg = ''):
      return ''.join(['Foo message is: ',str(foomsg)])

from arcom.service import Service

class DummyService(Service):
  """ DummyService class based on Service class in arcom.service
      Functionality of Dummy is accessible through this service."""
  def __init__(self, cfg):
    """ Constructor of the DummyService

        DummyService(cfg)

        'cfg' is an XMLNode which contains the config of this service."""

    # set service name
    self.service_name = 'Dummy'

    # set name of provided method
    self.request_names = ['foo','status']

    # call the Service's constructor
```

```
    Service.__init__(self,\
                     [{'request_names' : self.request_names,\
                       'namespace_prefix': 'dmmy',\
                       'namespace_uri': 'urn:foo'}],\
                     cfg)

    self.dummy = Dummy()

  def foo(self, inpayload):
    # get request
    req = inpayload.Child()
    fooreq = str(req)

    # get the job done
    foomsg = self.dummy.foo(fooreq)

    # create response
    out = self._new_soap_payload()
    response_node = out.NewChild('dmmy:fooResponse')
    response_node.Set(foomsg)

    return out

  def status(self, inpayload):
    # get local info
    dummyinfo = self.GetLocalInformation()

    # get service status
    dummystatus = str(dummyinfo.Get('AdminDomain').Get('Services').Get('Service').\
    Get('Endpoint').Get('ServingState'))

    # create response
    out = self._new_soap_payload()
    response_node = out.NewChild('dmmy:statusResponse')
    response_node.Set('Status: ' + dummystatus)

    return out
```

D arcom.threadpool test - threadpooltest.py

```
#!/usr/bin/python
# arcom.threadpool test

import threading, traceback

from time import sleep

from arcom.threadpool import ThreadPool, ThreadPoolThread, ReadWriteLock

COUNTER = 0

# waitTask: just sleep for a number of seconds
def waitTask(data):
  num = data[0]
  time = data[1]
  print "(%d): WaitTask starting" % num
  print "WaitTask sleeping for %d seconds" % time
```

```
  sleep(time)
  return "Waiter (%d)" % num

locker = ReadWriteLock()

# addTask: increment COUNTER with inc
def addTask(data):
  num = data[0]
  inc = data[1]
  global COUNTER
  nadds = 3
  for i in range(nadds):
    # re-acquiring lock for every add
    # don't do this at home...
    locker.acquire_write()
    COUNTER += inc
    print "(%d): Added %d to counter, counter is now %d"%(num,inc,COUNTER)
    locker.release_write()
    # sleep a bit to give other threads enough time to acquire the lock
    sleep(1)
  return "addTask", inc

# Both tasks use the same callback
def taskCallback(data):
  print "Callback called for", data

# Create a pool with three worker threads
pool = ThreadPool(3)

print "\nExample1 - waitTask"

# Insert tasks into the queue and let them run
pool.queueTask(waitTask, (1,4), taskCallback)
pool.queueTask(waitTask, (2,1), taskCallback)
pool.queueTask(waitTask, (3,6), taskCallback)
pool.queueTask(waitTask, (4,2), taskCallback)

pool.joinAll()

pool = ThreadPool(4)

print "\nExample2 - addTask"

pool.queueTask(addTask, (5,10), taskCallback)
pool.queueTask(addTask, (6,10), taskCallback)
pool.queueTask(addTask, (7,10), taskCallback)
pool.queueTask(addTask, (8,10), taskCallback)

print "COUNTER before join: ", COUNTER
# When all tasks are finished, allow the threads to terminate
pool.joinAll()
print "COUNTER after join: ", COUNTER
```

E Examples of 2.1.1

E.1 Handling a list of strings

(Example 1)

```
import arc
#create an empty list
sl = arc.StringList()
#append strings to the list
#size increases
sl.size()
sl.append('apple')
sl.size()
sl.append('banana')
sl.size()
sl.append('lemon')
sl.size()
sl.append('orange')
sl.size()
#list members
sl[0], sl[1], sl[2], sl[3]
#do some slicing
tmp = sl[:2]
#now tmp contains the first two strings
tmp.size()
tmp[0], tmp[1]
#do some more slicing
tmp = sl[1:]
#now tmp contains all the strings except the first
tmp.size()
tmp[0], tmp[1], tmp[2]
```

E.2 arc.StringStringMap

(Example 2)

```
import arc
#create an empty map
ssm = arc.StringStringMap()
#add mapping
ssm['key1'] = 'value1'
ssm['key2'] = 'value2'
#get keys
ssm.keys()
#get value for 'key1'
ssm['key1']
#get value for 'key2'
ssm['key2']
```

F Examples of 2.1.2

F.1 String operator - simple node

(Example 3)

```
import arc
#create an XMLNode
mynode = arc.XMLNode(arc.NS({'me':'http://example.com/myExample'}),'me:myNode')
#set text content
mynode.Set('Hello, World!')
#representation of the node
mynode.GetXML()
#String operator in action
str(mynode)
```

F.2 String operator - complex node

(Example 4)

```
# String operator of XMLNode works with the text content.
# It does not care about child nodes or text contents of those nodes.

# build a tree and set text content in nodes
#       r
#     / | \
#    /  |  \
#   /   |   \
# c01  c02  c03
#      /\    |
#     /  \  c06
#   c04  c05

# create the root node
r = arc.XMLNode(arc.NS(),'r')
# set text content
r.Set('R')
# create rest of the tree
c01 = r.NewChild('c01')
c01.Set('C-01')
c02 = r.NewChild('c02')
c02.Set('C-02')
c03 = r.NewChild('c03')
c03.Set('C-03')
c04 = c02.NewChild('c04')
c04.Set('C-04')
c05 = c02.NewChild('c05')
c05.Set('C-05')
c06 = c03.NewChild('c06')
c06.Set('C-06')
# String operator does not care about child nodes
str(c03)
str(c02)
str(r)
# XML representation of a complex node
c02.GetXML()
```

F.3 Occurence of out_xml_str as an output value

(Example 5)

```
import arc
#create node
```

```
n = arc.XMLNode(arc.NS({'me':'myNS'}),'me:myNode')
#set content
n.Set('Hello, World!')
#get document
#GetDoc according to API:
#void GetDoc(std::string &out_xml_str, bool user_friendly=false) const
#out_xml_str is turned into an output value and user_friendly is an optional
#parameter so there are no necessary parameters this time
#result will hold the output value
result = n.GetDoc()
#show result
result
```

F.4 Using LogStream to add new destination to the root logger

(Example 6)

```
import arc
import sys
#get root logger
root_logger = arc.Logger_getRootLogger()
#create a LogStream; sys.stdout would be OK
stream = arc.LogStream(sys.stdout)
#add destination to root logger
root_logger.addDestination(stream)
#log a message
#result immediately appears on sys.stdout
root_logger.msg(arc.INFO, 'Hello, World!')
```

G Examples of 2.1.3

G.1 MessageAttributes - getAll

(Example 7)

```
import arc
#create MessageAttributes object
mas = arc.MessageAttributes()
#add some key-value pairs
mas.add('key1', 'value1')
mas.add('key3', 'value3')
mas.add('key2', 'value2')
#get all attributes
all = mas.getAll()
#show result
all
#function returns a dictionary
type(all)
#show keys
all.keys()
#show values
all.values()
```

G.2 SOAPEnvelope - out_xml_str

(Example 8)

```
import arc
#create a namespace
ns = arc.NS({'me':'http://example.com/myExample'})
#create an empty SOAPEnvelope
#use the namespace created above
se = arc.SOAPEnvelope(ns,False)
#show it
se.GetXML()
```

H Examples of 2.1.4

H.1 ClientSOAP - process

(Example 9)

```
import arc
#create default config
cfg = arc.MCCConfig()
#create URL
url = arc.URL('http://localhost:50000/Echo')
#create payload
payload = arc.PayloadSOAP(arc.NS({'echo':'urn:echo'}))
#create payload content
#and set echo message
payload.NewChild('echo:echo').NewChild('echo:say').Set('Hello, World!')
#create client
client = arc.ClientSOAP(cfg,url)
#let the client call the service
response, status = client.process(payload)
#see response
print response.GetXML()
#check status
status.getKind() == arc.STATUS_OK
```

I Examples of 3.1

I.1 Importing the Logger class from module "logger" of "arcom" package

(Example 13)

```
import arcom
# From the logger module within the arcom package
# import the Logger class
Logger = arcom.import_class_from_string('arcom.logger.Logger')
# Now the class could be access through Logger
Logger
```

I.2 Getting attributes from an XMLNode

(Example 14)

```
import arc
import arcom
# Create XMLNode
n = arc.XMLNode(arc.NS({'me':'http://example.com/myExample'}),'me:myExample')
```

```
# Add attributes to node and set their values
n.NewAttribute('foo').Set('Hello')
n.NewAttribute('moo').Set('World')
# Show XML
n.GetXML()
# get attributes of this XMLNode
a = arcom.get_attributes(n)
# result is a dictionary
type(a)
# show dictionary
a
# extracting an attribute value
a['moo']
```

I.3 Getting child nodes

(Example 15)

```
import arc
import arcom
# Create root node
a = arc.XMLNode(arc.NS(),'a')
# Create child nodes for 'a'
b = a.NewChild('b')
c = a.NewChild('c')
d = a.NewChild('d')
# Create child nodes for 'c'
e = c.NewChild('e')
f = c.NewChild('f')
# Create child node for 'd'
g = d.NewChild('g')
# show XML
a.GetXML()
# get child nodes for 'a'
tmp = arcom.get_child_nodes(a)
# result is a list
type(tmp)
# 'a' has 3 children
len(tmp)
# show name for each
', '.join(x.Name() for x in tmp)
# 'b' has no children
tmp = arcom.get_child_nodes(b)
len(tmp)
# 'c' has 2 children: 'e' and 'f'
tmp = arcom.get_child_nodes(c)
len(tmp)
', '.join(x.Name() for x in tmp)
# 'd' has 1 child: 'g'
tmp = arcom.get_child_nodes(d)
len(tmp)
', '.join(x.Name() for x in tmp)
# nodes 'e', 'f' and 'g' have no children
tmp = arcom.get_child_nodes(e)
len(tmp)
tmp = arcom.get_child_nodes(f)
len(tmp)
tmp = arcom.get_child_nodes(g)
```

```
len(tmp)
```

I.4 Get values of specified children

(Example 16)

```
import arc
import arcom
# Create XMLNode
n = arc.XMLNode(arc.NS(),'node')
# Create 3 child nodes (two of which get the same name)
x = n.NewChild('same')
y = n.NewChild('same')
z = n.NewChild('different')
# Set values for nodes
x.Set('firstEQ')
y.Set('secondEQ')
z.Set('DIFF')
# Show XML
n.GetXML()
# Get child values where name is 'same'
tmp = arcom.get_child_values_by_name(n,'same')
# Result is a list
type(tmp)
# Show result
tmp
# Get child values where name is 'different'
tmp = arcom.get_child_values_by_name(n,'different')
# Show result
tmp
```

I.5 Creating DataPoint from URL

(Example 17)

```
import arc
import arcom
tmpList = []
status = ''
# create DataPoint from a local directory
dp = arcom.datapoint_from_url('file:///usr/local/share/arc')
# list files
(files, stat) = dp.ListFiles()
# if it is not empty
if files:
    status = 'found'
    # for all the entries, get type and name
    for f in files:
        if (f.GetType() == arc.FileInfo.file_type_file):
            type = 'file'
        elif (f.GetType() == arc.FileInfo.file_type_dir):
            type = 'dir'
        else:
            type = 'unknown'
        # get results together in a list
        tmpList.append(f.GetName() + ' (' + type + ')\n')
else:
```

```
    status = 'Could not access data. Reason: %s' % str(stat)

# see result
str(stat)
# show list
tmpList
```

I.6 Parsing a URL

(Example 18)

```
import arcom
proto, host, port, path = arcom.parse_url('boo://no.one.here:123/foo')
proto
host
port
path
```

J Examples of 3.2

J.1 Creating a client and calling the echo service (XMLTree)

(Example 19)

```
import arc
import arcom
# Import the Client class
Client = arcom.import_class_from_string('arcom.client.Client')
# Import the XMLTree class
XMLTree = arcom.import_class_from_string('arcom.xmltree.XMLTree')
# Create namespace - it will be used for the message sent
ns = arc.NS({'echo':'urn:echo'})
# Create client
c = Client('http://your.server.example.com:50000/Echo',ns,print_xml=True)
# Create message
msg = XMLTree(from_tree = ('echo:echo',[('echo:say', 'Hello, World!')]))
# Let the client do what it is meant for
# Note that we created the client with print_xml=True
# so both request and response will be displayed in an easy-to-read form
c.call(msg)
```

J.2 Creating a client and calling the echo service (SOAP)

(Example 20)

```
import arc
import arcom
# Import the Client class
Client = arcom.import_class_from_string('arcom.client.Client')
# Create namespace - it will be used for the message sent
ns = arc.NS({'echo':'urn:echo'})
# Create client
c = Client('http://arctest.ki.iif.hu:50000/Echo',ns,print_xml=True)
# Create SOAP Payload
pl = arc.PayloadSOAP(ns)
```

```
# Create message structure and set content
pl.NewChild('echo:echo',ns).NewChild('echo:say',ns).Set('Hello, World!')
# Let client do the call; see response
c.call_raw(pl)
```

K Examples of 3.3

K.1 Using the Logger

(Example 21)

```
import arc
import arcom
import sys
# Import get_logger function
from arcom.logger import get_logger
# Create logger
rl = get_logger()
rl.logger
arclogger = rl.logger
# Add new destination
arclogger.addDestination(arc.LogStream(sys.stdout))
# Log a message
# Note that the message is also returned as an output
rl.msg(arc.INFO, 'Hello, World!')
```

L Examples of 3.4

L.1 Creating policy

(Example 23)

```
from arcom.security import AuthPolicy
# create policy object
p = AuthPolicy()
# set policy for Alice and Bob
# Alice is allowed to read and add entries (addEntry)
# Bob is only allowed to read
AlicePol = ('Alice','+read +addEntry')
BobPol = ('Bob','+read')
p.set_policy([AlicePol,BobPol])
# see Policy document
print p.get_policy('ARCAuth')
```

L.2 Decision making

(Example 24)

```
dsa = DummySecAttr('Alice')
# set 'thisown' to False to avoid problems
dsa.thisown = False

# export XML in ARCAuth format
ex = dsa.Export()
```

```
from arcom.security import AuthPolicy
# create policy object
p = AuthPolicy()
# set policy for Alice
# Alice is allowed to read and add entries (addEntry)
AlicePol = ('Alice','+read +addEntry')
p.set_policy([AlicePol])

from arcom.security import make_decision

# get policy XML
px = p.get_policy()
rx = ex.GetXML()

decision = make_decision(px, rx)

# Possible results:
# arc.DECISION_PERMIT -- 0
# arc.DECISION_DENY -- 1
# arc.DECISION_INDETERMINATE -- 2
# arc.DECISION_NOT_APPLICABLE -- 3

decision
```

L.3 SSL config example

(Example 25)

```
import arc
import arcom
from arcom.security import parse_ssl_config

xml_str = file('/etc/arc/ssl_config_example.xml').read()
configNode = arc.XMLNode(xml_str)

serviceNode = configNode.Get('Chain').Get('Service')

print parse_ssl_config(serviceNode)
```

M Examples of 3.5

M.1 Using DummyService - an example service based on arcom.service.Service

(Example 26)

```
# Reminder:
# import arc
#
# class Dummy:
#   def foo(self, foomsg = ''):
#     return ''.join(['Foo message is: ',str(foomsg)])
#
# from arcom.service import Service
#
# class DummyService(Service):
#   """ DummyService class based on Service class in arcom.service """
```

```
#   ...

# Create instance of DummyService
ds = DummyService(None)
# Create namespace - it will be used for the message sent
ns = arc.NS({'dmmy':'urn:foo'})
# Create SOAP Envelope
se = arc.SOAPEnvelope(ns)
# Create content
se.NewChild('dmmy:foo').Set('Foo!')
# Create SOAP Message
msg = arc.SOAPMessage()
# Payload
pls = arc.PayloadSOAP(se)
# Set Message Payload
msg.Payload(pls)
# create outgoing message that will hold response
outse = arc.SOAPEnvelope(ns)
outmsg = arc.SOAPMessage()
outpls = arc.PayloadSOAP(outse)
outmsg.Payload(outpls)
# call service
ds.process(msg, outmsg)
# show result
print outmsg.Payload().GetXML()
```

M.2 Retrieving state of DummyService

(Example 27)

```
# Reminder:
# import arc
#
# ...
#
# from arcom.service import Service
#
# class DummyService(Service):
#   """ DummyService class based on Service class in arcom.service """
#
#   ...
#
#   def status(self, inpayload):
#       # get local info
#       fooinfo = self.GetLocalInformation()
#
#       # get service status
#       foostatus = str(fooinfo.Get('AdminDomain').Get('Services').Get('Service').\
#       Get('Endpoint').Get('ServingState'))
#       ...

# Create instance of DummyService
ds = DummyService(None)
# Create namespace - it will be used for the message sent
ns = arc.NS({'dmmy':'urn:foo'})
# Create SOAP Envelope
se = arc.SOAPEnvelope(ns)
# Create content
```

```
se.NewChild('dmmy:status')
# Create SOAP Message
msg = arc.SOAPMessage()
# Payload
pls = arc.PayloadSOAP(se)
# Set Message Payload
msg.Payload(pls)
# create outgoing message that will hold response
outse = arc.SOAPEnvelope(ns)
outmsg = arc.SOAPMessage()
outpls = arc.PayloadSOAP(outse)
outmsg.Payload(outpls)
# call service
ds.process(msg, outmsg)
# show result
print outmsg.Payload().GetXML()
```

M.3 parse_node

(Example 28)

```
from arc import XMLNode
from arcom.service import parse_node

# create xml node
xml = XMLNode('''
<myList>
  <myElement>
    <myID>0</myID>
    <LN>/</LN>
  </myElement>
  <myElement>
    <myID>1</myID>
    <LN>/testfile</LN>
  </myElement>
</myList>''')

# default: single = False and string = True
parse_node(xml, ['myID','LN'])
# single = True
parse_node(xml, ['myID','LN'], single = True)
parse_node(xml, ['myID','LN'], True)
# string = False
parse_node(xml, ['myID','LN'], string = False)
```

M.4 parse_to_dict

(Example 29)

```
from arc import XMLNode
from arcom.service import parse_to_dict

# create xml node
xml = XMLNode('''
<myList>
  <myElement>
    <myID>123</myID>
```

```
    <refID>abc</refID>
    <state>alive</state>
    <size>123456</size>
  </myElement>
  <myElement>
    <myID>456</myID>
    <refID>fed</refID>
    <state>alive</state>
    <size>987</size>
  </myElement>
</myList>''')

# example 1 - 'myID' as key
# 'state' and 'size' goes to the inner dictionary
parse_to_dict(xml, ['myID','state','size'])

# example 2 - 'myID' as key
# 'myID' also goes to the inner dictionary
parse_to_dict(xml, ['myID','myID','state','size'])
```

M.5 create_response

(Example 30)

```
import arc
from arcom.service import create_response

method_name = 'dummy'
# create tag_names; 'myID' will be the key
tag_names = ['myID', 'state', 'size']
# create elements
# elements of the lists in this dictionary are values and will be tagged
# according to names found in tag_names except for the first element
# e.g.: 'alive' will be tagged with 'state'
elements = {'123': ['alive', '123456'], '456': ['alive', '987']}
# create empty payload
payload = arc.PayloadSOAP(arc.NS())

# create response
response = create_response(method_name, tag_names, elements, payload)

# show it
print response.GetXML(True)
```

M.6 node_to_data - 1

(Example 31)

```
from arc import XMLNode
from arcom.service import node_to_data

# create node
xml = XMLNode('''
<myElement>
  <myID>0</myID>
  <refID>abc</refID>
  <section>states</section>
```

```
  <property>spares</property>
  <value>2</value>
  <dummy>foo</dummy>
  <changeType>set</changeType>
</myElement>
''')

# create names
# 'myID' will be the key
# all the other names will be in a list in the specified order
# 'dummy' will be left out
names = ['myID', 'refID', 'changeType', 'section', 'property', 'value']

# call node_to_data
node_to_data(xml,names)
```

M.7 node_to_data - 2

(Example 32)

```
from arc import XMLNode
from arcom.service import node_to_data

# create node
xml = XMLNode('''
<getRequest>
  <GUID>11</GUID>
  <myID>99</myID>
</getRequest>
''')

# create names
# 'myID' will be the key
names = ['myID', 'GUID']

# call node_to_data
# GUID is the single value so do not put it in a list
node_to_data(xml,names, True)
```

M.8 get_data_node

(Example 33)

```
from arc import XMLNode
from arcom.service import get_data_node

# create node
xml = XMLNode('''
<Envelope>
  <Body>
    <method>
      <data>Value</data>
    </method>
  </Body>
</Envelope>
''')
```

```
# call get_data_node
dn = get_data_node(xml)

# show result
dn.GetXML()
```

N Examples of 3.6

N.1 arcom.threadpool

(Example 34)

```
$./threadpooltest.py
```

O Examples of 3.7

O.1 Creating an XMLTree

(Example 35)

```
from arc import XMLNode
from arcom.xmltree import XMLTree

# create XMLTree sources
# xmlstr - will be used to create an XMLTree from string
# xmlnodestr - will be used to create 'node' (an XMLNode)
# node - will be used to create an XMLTree from XMLNode
# treestruct - for creating XMLTree from tree structure
xmlstr = '<echo><say>Hello!</say></echo>'
xmlnodestr = '<node><subnode>Subnode</subnode></node>'
node = XMLNode(xmlnodestr)
treestruct = ('root', [('leaf','Leaf')])

# from_tree has the highest priority
t1 = XMLTree(from_node = node, from_string = xmlstr, from_tree = treestruct)
t1.get()

# no from_tree supplied; from_node will be used
t2 = XMLTree(from_node = node, from_string = xmlstr)
t2.get()

# from_string only
t3 = XMLTree(from_string = xmlstr)
t3.get()
```

O.2 XMLTree - forget namespace

(Example 36)

```
from arc import XMLNode
from arcom.xmltree import XMLTree

# Create XMLTree objects to demonstrate
# the use of forget namespace feature
# node - is an XMLNode
```

```
# tree - is an XMLTree that is created from 'node'
node = XMLNode('<soap-env:Envelope xmlns:hash="urn:hash" \
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">\
<soap-env:Body>\
<hash:get>\
<hash:IDs>\
<hash:ID>0</hash:ID>\
<hash:ID>1</hash:ID>\
<hash:ID>2</hash:ID>\
</hash:IDs>\
</hash:get>\
</soap-env:Body>\
</soap-env:Envelope>')

tree = XMLTree(node)
# show tree structure
print tree

# forget_namespace is ignored when from_tree is used
tree2 = XMLTree(from_tree = tree, forget_namespace = True)
print tree2

# forget_namespace with from_node
tree3 = XMLTree(from_node = node, forget_namespace = True)
# it works as expected
print tree3

# forget_namespace with from_string
strnode = '<myns:foo xmlns:myns="urn:foo"><myns:dummy>Hello, World!</myns:dummy>\
</myns:foo>'
tree4 = XMLTree(from_string = strnode, forget_namespace = True)
# it also works
print tree4
```

O.3 XMLTree - rewrite

(Example 37)

```
from arc import XMLNode
from arcom.xmltree import XMLTree

# Create XMLTree objects to demonstrate
# the use of rewrite feature
strnode = '<myns:foo xmlns:myns="urn:foo"><myns:dummy>Hello, World!</myns:dummy>\
</myns:foo>'
tree = XMLTree(from_string = strnode,\
               rewrite = {'myns:foo':'myns:boo'},\
               forget_namespace = False)

# 'myns:foo' is turned into 'myns:boo'
print tree

# combine with forget_namespace
tree2 = XMLTree(from_string = strnode,\
                rewrite = {'foo':'boo'},\
```

```
                forget_namespace = True)
# 'forget_namespace' is True so namespace prefix 'myns' is being stripped off
# ('myns:foo' -> 'foo'; 'myns:dummy' -> 'dummy')
# then 'foo' is turned into 'boo'
print tree2
```

O.4 XMLTree - add_to_node

(Example 38)

```
from arc import XMLNode
from arcom.xmltree import XMLTree

node = XMLNode('<node/>')

# create XMLTree
tree = XMLTree(from_string = '<root><dummy/><hello>World</hello></root>')

# from 'tree' add 'dummy' to 'node'
tree.add_to_node(node, '/root/dummy')

# show result
print node.GetXML(True)

# from 'tree' add 'hello' to 'dummy' in 'node'
dummy = node.Get('dummy')
tree.add_to_node(dummy, '/root/hello')

# show result
print node.GetXML(True)
```

O.5 XMLTree - pretty_xml

(Example 39)

```
from arcom.xmltree import XMLTree

# create XMLTree
tree = XMLTree(from_string = '<root><Hello>World</Hello></root>')

# select entire tree
# indent children with space character
# apply prefix for every new line
print tree.pretty_xml(indent=' ', path='', prefix='# ')

# note that it is possible to select multiple parts
# create an example XMLTree to demonstrate that
tree2 = XMLTree(from_string = '<root><sub><Hello>World</Hello></sub>\
<sub><dummy>foo</dummy></sub></root>')

# show '/root/sub'
print tree2.pretty_xml(indent=' ', path='/root/sub', prefix='# ')
```

O.6 XMLTree - __str__

(Example 40)

```
from arcom.xmltree import XMLTree

# create XMLTree
tree = XMLTree(from_string = '<root><dummy>foo</dummy></root>')

# show tree
# print will result in __str__ being called
# when trying to convert XMLTree to string
print tree
```

O.7 XMLTree - get

(Example 41)

```
from arcom.xmltree import XMLTree

# create tree structure
tree = \
('root',\
  [('trunk',\
    [('leaf','1'),\
     ('leaf','2')]\
  )]\
)

# create XMLTree from tree structure
xt = XMLTree(from_tree = tree)

# if called with path being None
# get() selects the root node
xt.get()

# Path is just a plain path.
# Empty tag name matches everything, so - in this particular case -
# all expressions below will produce the same result.
xt.get('/root/trunk/leaf')
xt.get('//trunk/leaf')
xt.get('/root//leaf')
xt.get('/root/trunk/')
xt.get('///leaf')
xt.get('///')
```

O.8 XMLTree - get_trees

(Example 42)

```
from arcom.xmltree import XMLTree

# create tree structure
tree = \
('trunk',\
  [( 'branch',\
     [('leaf','1'),\
      ('leaf','2')]\
   ),\
   ( 'branch',\
     [('leaf','3')]\
```

```
   )]\
)

# create XMLTree from tree structure
xt = XMLTree(from_tree = tree)

# get branches ('/trunk/branch') in a list
blist = xt.get_trees('/trunk/branch')

# show branches in blist
for b in blist:
    print b

# get leaves ('/trunk/branch/leaf') in a list
llist = xt.get_trees('/trunk/branch/leaf')

# show leaves in llist
for l in llist:
    print l
```

O.9 XMLTree - get_value

(Example 43)

```
from arcom.xmltree import XMLTree

# create tree structure
ts = ('branch',[('leaf','1')])

# create XMLTree from tree structure
tree = XMLTree(from_tree = ts)

# get_value for '/branch/leaf'
# will return the value from 'leaf'
tree.get_value('/branch/leaf')

# get_value for '/branch'
# will return the value from 'branch'
# ie the list of its children
tree.get_value('/branch')

# get_value for a path that does not match
# provide a default value of 'N/A'
tree.get_value('/dummy','N/A')
```

O.10 XMLTree - add_tree

(Example 44)

```
from arcom.xmltree import XMLTree

# create tree structure
ts = \
('root',\
  [ ('branch',\
      [ ('leaf','1'),\
```

```
        ('leaf','2')\
      ]\
    ),\
    ('branch',\
      [ ('leaf','3'),\
        ('leaf','4')\
      ]\
    )\
  ]\
)

# create XMLTree from tree structure
tree = XMLTree(from_tree = ts)

# add a new leaf ('leaf','5') to '/root/branch'
# the new leaf will be added to the first node
# matched by path ('/root/branch')
tree.add_tree(('leaf','5'), '/root/branch')

# show tree
print tree
```

O.11 XMLTree - get_values

(Example 45)

```
from arcom.xmltree import XMLTree

# create tree structure
ts = \
('root',\
  [ ('branch',\
      [ ('leaf','1'),\
        ('leaf','2')\
      ]\
    ),\
    ('branch',\
      [ ('leaf','3'),\
        ('leaf','4')\
      ]\
    )\
  ]\
)

# create XMLTree from tree structure
tree = XMLTree(from_tree = ts)

# get values of branches ('/root/branch') in a list
# this will return the lists of children of the branches
tree.get_values('/root/branch')

# get values of leaves ('/root/branch/leaf') in a list
# this will return the values of leaves
tree.get_values('/root/branch/leaf')

# try a path that does not match
# this will return an empty list
tree.get_values('/dummy')
```

O.12 XMLTree - get_dict

(Example 46)

```
from arcom.xmltree import XMLTree

# create tree structure
ts =\
('person',\
  [ ('name', 'Alice'),\
    ('id', '11'),\
    ('job', 'Librarian')\
  ]
)

# create XMLTree from tree structure
tree = XMLTree(from_tree = ts)

# get dictionary from '/person'
# only name and job is to be returned
tree.get_dict('/person',{'name':'name','job':'job'})
```

O.13 XMLTree - get_dicts

(Example 47)

```
from arcom.xmltree import XMLTree

# create tree structure
ts =\
('persons',\
  [ ('person',\
      [ ('name', 'Alice'),\
        ('id', '11'),\
        ('job', 'Librarian')\
      ]\
    ),\
    ('person',\
      [ ('name', 'Bob'),\
        ('id', '12'),\
        ('job', 'Bartender')\
      ]\
    )\
  ]\
)

# create XMLTree from tree structure
tree = XMLTree(from_tree = ts)

# get dictionaries for '/persons/person'
tree.get_dicts('/persons/person')
```