PyOPC

A Python Framework for the OPC XML-DA 1.0 Standard

Hermann Himmelbauer

Klosterneuburg, the November 7, 2006

Contents

1 Introduction

The Open Linking and Embedding for Process Control (OPC) consortium released several open standards, which address interfaces for vertical integration in industrial automation. These standards can be used to build Internet/fieldbus interfaces which are placed on gateway devices as shown in figure 1

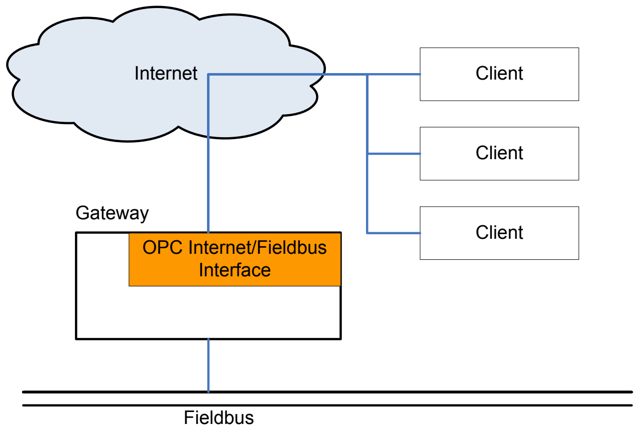

Figure 1: Internet/fieldbus interface on a gateway

Historically, OPC used the Distributed Component Object Model (DCOM) for the underlying communication technology. DCOM has the disadvantage of being platform specific: it is only available for Microsoft Windows based systems. Other platforms, such as Linux, can therefore not retrieve fieldbus data from DCOM based servers. Another disadvantage of DCOM is that it can not easily bypass firewalls, hence access will often be limited to certain segments of a corporate network.

In the last years, a new technology, called SOAP Web services, emerged. [LIV02] defines a Web Service as: "a method or function that is available for other applications to access over the Internet.". Web services enable Remote Procedure Calls (RPC) and have the following key features:

High level of interoperability: Web services technologies are all based on strictly defined open standards[1].

High networking abilities: As an underlying communication protocol, Web services utilize Internet protocols such as the Hyper Text Transfer Protocol (HTTP) or the Simple Mail Transfer Protocol (SMTP). These protocols have high networking abilities and may moreover penetrate firewalls.

Protocol legible by humans: The Simple Object Access Protocol (SOAP)[2] is based on the Extended Markup Language (XML), which is legible to humans. This way, testing and debugging of Web services is far easier than with binary protocols.

Documentation: Another underlying technology of SOAP is the "Web Service Definition Language" (WSDL) which may be used to define the service, especially by constraining

[1]Most technologies are W3C standards.

[2]To be correct, SOAP is not an acronym for Simple Object Access Protocol anymore, instead it's simply "SOAP".

the format of the SOAP protocol. WSDL utilizes the XML Schema language for defining these SOAP messages[3]. These WSDL documents can be utilized by frameworks to generate stubs that provide a base for accessing a Web service.

Validation: WSDL in combination with a validating XML parser enable the validation of SOAP messages. This way, custom code will never receive syntactically or semantically erroneous data, which should improve the stability of the service.

SOAP Web services are seen as a successor to several alternative technologies such as DCOM and are already broadly accepted by the industry. More information about the SOAP protocol can be found in [SEE02] and [LIV02].

The OPC consortium reacted on this technological evolution by adopting SOAP Web services for their standards. One recent addition of OPC is the "XML Data Access Version 1.0" (XML-DA 1.0) standard. This standard deals with access of underlying fieldbus technologies and covers the following aspects:

Information model: The specification provides a simple information model, based on "OPC Items" which represent a piece of information, similar to fieldbus data points. These items can be arranged hierarchically.

Data types: OPC XML-DA adopts several XML-Schema based data types, such as integer, float, date/time specific types. Moreover it defines arrays which are based on these basic types.

Operations: The standard specifies 8 operations such as reading/writing and browsing which can be used to access the underlying fieldbus.

Subscription: The specification further introduces a mechanism to retrieve only changed items, called "Subscription". Clients may thus subscribe to items and use a dedicated polling operation to retrieve changed data.

The standard does not address security, instead it relies on underlying Web service technologies[4]. More information about OPC XML-DA can be found in [HIM06] and [OPCXMLDA].

Although OPC XML-DA is based on open and standardized technologies, it can nevertheless be tedious to build services based on this standard. Therefore several OPC frameworks are available that introduce simple building of client and server applications. However, most of these frameworks are not freely available, moreover most of them are based on Microsoft's .Net framework and are therefore platform dependent.

Due to these limitations, the PyOPC framework was developed, which fully implements the OPC XML-DA standard, enabling developers to build OPC XML-DA based applications in an easy way.

[3]More information about XML Schema can be found in [VLI02].

[4]For instance, HTTPS can be used to secure a HTTP channel.

2 Installation/Quickstart

Before installing the PyOPC framework, the following three software packages have to be installed:

Python: The Python programming language can be downloaded from http://www.python.org. It is available for a variety of operating systems. **The Python version must be at least 2.4.**

Zolera Soap Infrastructure (ZSI): The ZSI framework is used for parsing and generating the SOAP messages. It is available from http://pywebsvcs.sourceforge.net/. ZSI is still under development, therefore different releases may not work. The ZSI-2.0_rc3 release is known to work with PyOPC.

Twisted: The Twisted server framework is used for the server functionality. It is available from http://twistedmatrix.com/. All recently released versions should be appropriate.

Installation instructions for the above software packages should be available at the given websites.

PyOPC does currently not have an installer but is nevertheless relatively easy to install. At first the PyOPC package has to be decompressed. Then Python has to be informed where to find it. This is done by adding the location of the PyOPC variable to the environment variable *PYTHONPATH*.

If everything is installed correctly, the next step may be to test the installation. PyOPC contains extensive unit tests in the subdirectory “test”. These tests can either be run altogether by executing “runtests.sh” or selectively via the “trial” command from the Twisted framework. Hopefully, all tests will pass[5].

If all goes well, PyOPC is ready to use. As a quickstart, an existing OPC XML-DA server may be queried directly from python such as shown in listing 1[6].

```
from PyOPC.XDAClient import XDAClient

address='http://path/to/server'
xda = XDAClient(OPCServerAddress=address)
xda.GetStatus()
```

Listing 1: Accessing a remote OPC XML-DA server

[5]On some slower machines, certain server operations may fail as they rely on a predefined execution time of certain operations.

[6]A demo server, implemented with PyOPC, is set up at http://violin.qwer.tk:8000/, which may, however, not be available all the time. There are several other public OPC XML-DA compliant servers available. Some addresses for such servers may be found at http://www.opcfoundation.org, moreover Advosol also offers access to some demo servers (see http://www.advosol.com).

3 Architecture and Basic Concepts

The PyOPC framework supports the rapid development of OPC XML-DA compliant clients and servers and provides the following features:

Open source: PyOPC and all underlying technologies are open source projects.

Multi-platform capable: The underlying programming language of PyOPC is Python, which is available on most platforms, such as Microsoft Windows, Linux, Mac OS X and others. Applications built with PyOPC can be run on all platforms with Python support. A good introduction to the Python programming language can be found in [PIL05] and [MAR03].

Ease of use: Various complex functionality of the OPC XML-DA specification is automatically handled by the PyOPC framework, therefore the developer does not need to cope with it. Nevertheless, the programmer may also choose to override this functionality and thus implement it in his own way.

Extensible and reusable: Basic framework functionality can be extended by the programmer, moreover, OPC servers built with PyOPC can be added to the framework as a custom libraries, which can then be reused by other applications.

An OPC XML-DA compliant framework needs to support several technologies, especially building servers which process concurrent requests and handle the SOAP and HTTP protocol. Although Python has an extensive collection of libraries, it does not fulfill these requirements. Therefore two additional Python frameworks are used, which have to be available on systems that provide PyOPC based applications. These basic technologies are illustrated in figure 2.

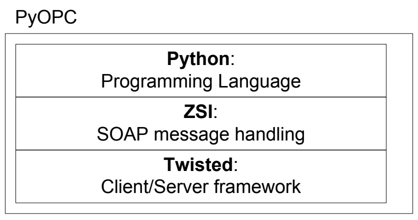

Figure 2: Underlying Technologies of the PyOPC Framework

Zolera Soap Infrastructure (ZSI): This SOAP framework enables parsing and serializing SOAP messages. PyOPC uses ZSI to read and create OPC XML-DA compliant SOAP messages[7].

Twisted: Twisted is an asynchronous client/server framework. It implements a variety of Internet protocols, such as HTTP and SMTP and uses an event based mechanism to enable the development of clients and servers which can handle concurrent requests.

In order to develop applications with Twisted, the programmer has to lay out his program according to this event-based architecture. Therefore building complex server applications with PyOPC require some understanding of the basic concepts of Twisted. More information about this framework can be found at [FET06].

[7]This process is hidden from developers, instead PyOPC provides abstract Python objects for accessing underlying SOAP messages.

3.1 Basic PyOPC Architecture

The PyOPC framework supports the development of OPC XML-DA client and server applications. The OPC XML-DA standard defines the following eight operations, which can be used to access OPC data:

GetStatus: This operation is used to retrieve status information of the OPC server.

Read/Write: These operations are used to read and write OPC items[8].

Subscribe / SubscriptionPolledRequest / SubscriptionCancel: These operations are used to handle OPC subscriptions.

Browse: This operation is used to browse OPC items.

GetProperties: OPC items have so-called properties, which define and describe the item. GetProperties can be used to retrieve these OPC item properties.

Associated to each of these operation is a request and response SOAP message, resulting in 16 different messages. With ZSI, these messages can be parsed and serialized through specific Python objects, so-called "Typecodes". Although accessing typecodes is far easier than accessing the SOAP message itself, it is still a tedious task. Therefore PyOPC hides this process from the developer by defining various methods, which automatically handle the ZSI typecodes.

PyOPC introduces several classes, which are used throughout the framework. These classes inherit from each other, forming a class hierarchy, as illustrated in figure 3.

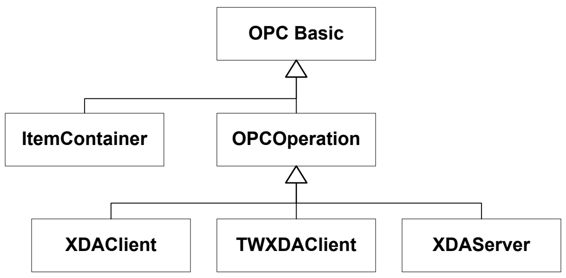

Figure 3: Class Hierarchy of the PyOPC Framework

The classes of this hierarchy implement the following functionality:

OPC Basic: This class implements basic issues that are used by other inheriting classes. Apart from several utility methods, the class defines all OPC errors.

OPCOperation: The OPCOperation class handles the generation and parsing of all OPC XML-DA SOAP messages. The class implements two read and write methods for each operation[9], which are automatically utilized for handling ZSI typecodes.

ItemContainer: This class represents OPC items and is described in detail below.

XDAClient: Simple OPC XML-DA clients may be built with this class.

[8]OPC items are basically containers which may hold a piece of information. They resemble fieldbus data points and may contain arbitrary data.

[9]Every request and response SOAP message has each a read and write method.

TWXDAClient: This class is an advanced way to build OPC XML-DA compliant clients. The TWXDAClient class utilizes Twisted to enable multiple, concurrent client requests.

XDAServer: The XDAServer class is the base to implement OPC XML-DA compliant servers with the PyOPC framework.

3.2 Representation of OPC XML-DA Data with Python Objects

As already mentioned, handling SOAP messages with ZSI is not simple. Therefore specific objects are defined by PyOPC which can be easily accessed and represent the corresponding SOAP messages.

OPC XML-DA compliant SOAP messages contain various options which may either concern the whole operation (global options) or may be item specific (local options). After closely examining these messages, it was found that these options can be mapped to two certain Python objects as depicted in figure 4.

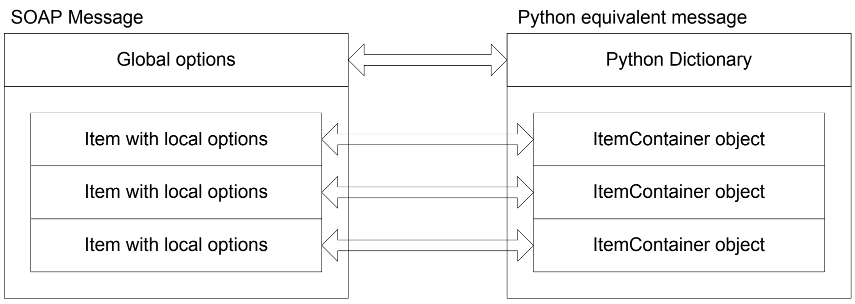

Figure 4: Python Objects Representing OPC XML-DA Messages

These two Python objects are the basic containers in the PyOPC framework and are used to transport OPC data. Therefore they are used as parameters for methods which represent OPC XML-DA operations. Listing 2 shows an example of an OPC XML-DA read operation.

```
1 item = ItemContainer(ItemName='test_name', MaxAge=500)
  item_list, global_options = xda.Read([item], ItemPath='test_path')
```

Listing 2: Usage of Python objects for representing global and local options of a OPC XML-DA compliant SOAP message

In line 1, an ItemContainer object is created, which contains the item specific (local) options “ItemName” and “MaxAge”. In line 2 the read operation is taking place, with a list of ItemContainer objects as the first parameter and the global option “ItemPath” as the second. The results of the read operation also consist of a list of ItemContainer objects and a Python dictionary, containing the local and global options.

Global and local options may sometimes be the same. In this case, the local options override the global ones. For instance, if the option “ItemPath” is specified globally and locally, the local, item specific ItemPath will have precedence.

As this mechanism is the same for all operations, it is sometimes possible to set options that have no meaning for the current operation. However, only relevant options will be included in the resulting SOAP message, other options are ignored by the PyOPC framework. For instance, setting the option “MaxAge” for an OPC Write operation has no effect at all.

There are a number of options that can be set for OPC XML-DA operations, which can be looked up in [OPCXMLDA]. These options are case sensitive, for instance “MaxAge” and “maxage” are handled as a different option.

Global OPC XML-DA options

Python dictionaries are used to represent global options. As denoted above, these dictionaries can then be used as parameters for OPC operations. However, often it is handy to define default global options that are used all over the application. For instance, it may make sense to define a server-specific default "MaxAge" option, which is automatically assigned to all according SOAP messages.

Therefore PyOPC allows to specify these global options at the creation of the client or server object, such as shown in listing 3. These global options will then be used for all operations, unless they are overridden by global options in the method parameters.

```
xda = XDAClient(ReturnErrorText=False,
                ReturnItemName=True,
                ReturnDiagnosticInfo=True,
                ItemPath='')
```

Listing 3: Assigning Global Options to a PyOPC Client Instance

The PyOPC framework always checks if global options, which are passed as method parameters, can be applied to the current OPC operation. If an option is unknown or misspelled, a Python *TypeError* will be raised. This way, errors due to accidental misspellings or improper use of options are prevented.

The ItemContainer Object

The ItemContainer object is used to store all local, item-specific OPC XML-DA data, such as the item value and item-specific local options. This information is stored in the object as object attributes and may be accessed as shown in listing 4.

```
  from PyOPC.OPCContainers import *
2 item = ItemContainer(ItemName='test_name',
                       MaxAge=500)
4 item.ItemName='other_name'
  maxage = item.MaxAge
```

Listing 4: Accessing the PyOPC ItemContainer Object

At the beginning of the example listing, the appropriate Python module is imported that contains the ItemContainer class. In line 2, item specific options are set at initialization time of the ItemContainer object, while line 4 and 5 show how to directly set and retrieve object attributes.

[OPCXMLDA] defines numerous, sometimes complex local options. Therefore errors due to misspellings can easily happen. To prevent such errors, the ItemContainer class defines all possible options via class attributes. In addition, an ItemContainer object will not allow the setting of undefined object attributes. If the developer accidentally tries to set a misspelled or unknown attribute, the PyOPC framework raises a Python AttributeError.

Qualified Names (QNames) and Namespaces

The OPC XML-DA specification and its associated SOAP messages sometimes contain *Qualified Names* (QNames). QNames consist of a namespace, most often in the form of an Uniform Resource Locator (URL) and a name. For this purpose, PyOPC defines a simple *QName* object, which is similar to a Python tuple.

Moreover PyOPC defines in the module *utils* the following global variables which may be used as the namespace part for QNames:

- *NS_XSD*, the namespace for XML-Schema
- *NS_ZSI*, the ZSI namespace
- *NS_XDA*, the namespace of the OPC XML-DA specification
- *NS_PYO*, the PyOPC namespace

An example how to create and access such a QName object is given in example 5, showing the creation of a predefined and a custom QName in line 3 and 4 and accessing parts of the QName in line 5:

```
1 from PyOPC.utils import *

3 qn1 = QName(NS\_XSD, 'string')
  qn2 = QName('http://my/name/space', 'test123')
5 url, name = qn2.URI, qn2.name
```

Listing 5: Handling Qualified Names (QNames) with PyOPC

OPC Item Properties

Every OPC Item may have so-called *properties* that contain further information about the item. Example properties would be the access rights or a description of the item. These OPC properties are modeled as a specific Python object, called *OPCProperty*, which contains the following information:

Name: The name uniquely identifies an OPC property. Names must be of the type *QName*.

Value: Properties most often have a value, for instance in case of a property "accessRights", it stores the strings *readable* or *writable*.

Description: In order to easily understand the meaning of a property, it can store a description.

ItemPath/ItemName: The address of the property, consisting of the ItemPath and ItemName.

ResultID/ErrorText: In case a property is erroneous, for instance if it cannot be read or does not exist, the error can be stored in a ResultID and a descriptive error text.

Listing 6 shows in line 1 to 5 how to create and access a PyOPC property object.

```
1 p1 = OPCProperty(Name = QName(NS_XDA, 'accessRights'),
                   Value = 'readable',
3                  Description = 'Access␣Rights')
  print p1.Name, p1.Value
5 p1.ItemPath = 'MyPath'

7 p2 = OPCProperty(Name = 'accessRights')
  print p2.Description
```

Listing 6: Creating and Accessing PyOPC Properties

The OPC XML-DA standard specifies various common properties, which should be preferred over custom properties, if possible. A full list of these available properties is given in [OPCXMLDA]. The alternative are custom properties that will often be in the namespace of PyOPC. Therefore the framework offers a simple shortcut in creating properties: if the property name is a string instead of a QName, PyOPC searches in a table for a matching OPC property. If one is found, the OPC XML-DA namespace is used, moreover the description is filled out automatically. If the property is unknown, the PyOPC namespace will automatically be used. This behavior is reflected in line 7 and 8 in listing 6.

Properties will be associated with items, therefore an ItemContainer object provides the following methods to add, delete and list properties:

- *addProperty(self, property) / addProperties(self,properties)* adds one property or a list of properties to the ItemContainer object.
- *getProperty(self, name)* retrieves a property according to its name
- *delProperty(self, name) / popProperty (self, name)* deletes a property. *popProperty* returns the property before deletion.
- *listProperties(self)* returns a list of all item properties.

Representation of the Item Value with PyOPC Data Types

OPC items have a *value*, containing the actual information, which corresponds to the value of a fieldbus data point. This value will be of a certain OPC data type, such as string or integer but also more complex types, such as an array. In order to access this information, PyOPC has to map the OPC data type to a corresponding Python data type, resulting in a possible data conversion. Table 1 describes all possible data conversions.

OPC to Python

OPC data type	**Python data type**
string	string
boolean	boolean
float	float
double	
decimal	decimal.decimal
long	long
int	
short	
byte	
unsignedLong	
unsingedInt	
unsignedShort	
unsignedByte	
base64Binary	string
dateTime	datetime.datetime
time	datetime.time
date	datetime.date
duration	datetime.timedelta
Qname	PyOPC Qname

Pyhon to OPC

Python data type	OPC data type
string	string
boolean	boolean
float	double
decimal.decimal	decimal
long/int	long
datetime.datetime	dateTime
datetime.time	time
datetime.date	date
datetime.timedelta	duration
PyOPC Qname	Qname

Table 1: Data Conversion Between OPC XML-DA and PyOPC

Currently, the following data types are not supported by the PyOPC framework[10]:

- The *decimal* data type is currently not supported at all and cannot be used.

[10]The reason for these unsupported data types is that the underlying SOAP framework, ZSI, does also not support them.

- The time based data types are not fully supported. Instead of using Pythons datetime module, all data types are converted to the Python time type. The duration type cannot be used. During conversion, fractions of seconds and the time zone is lost.
- The base64Binary type is mapped to a Python string. It may be reasonable to use a corresponding Python type instead, however this is currently not supported.

[OPCXMLDA] further defines arrays, which may contain OPC data types from the above. These arrays are directly mapped to Python lists[11] and its elements are converted as shown in Table 1.

3.3 Error Handling

OPC XML-DA describes the following two basic error types:

- OPC item specific, denoting that an OPC item is not accessible or is unknown
- OPC operation specific, identifying errors that concern the whole operation

These two errors are handled entirely different in both OPC XML-DA and PyOPC.

OPC Item Specific Errors

If an OPC item cannot be read, written, is unknown or is in any other way erroneous, the OPC server has to inform the client. These errors do not regard the whole operation, instead the response message, which transports the OPC items, implements the two following options that denote the item specific error:

ResultID: Item specific errors are always outlined by the ResultID. The OPC XML-DA specification distinguishes between so-called error and success codes, denoting if the transported item data is valid or not. The ResultID is of the type *QName* and contains a unique ID of the error. [OPCXMLDA] defines various errors, which are in the format of *E_FAIL* or *E_ACCESS_DENIED*. A complete description of these error and success codes can be found in [OPCXMLDA].

In case the provided OPC errors do not suffice, custom errors can be defined. PyOPC defines a few errors in its namespace, which can also be utilized.

ErrorText: This non-mandatory option may provide descriptive error text, which will make the reason of an error understandable to humans.

Despite of an item specific error, the response message may transport invalid item data. Therefore OPC XML-DA applications always have to check for the ResultID, so that item specific errors are detected.

[11]OPC XML-DA further allows to nest arrays in arrays, however this is currently not supported by PyOPC

OPC Operation Specific Errors

If the whole operation fails, for instance if the server is busy or malfunctioning, or if the client request is badly formatted, the server responds with a SOAP Fault message, which contains a detailed error description.

SOAP Faults are caught by the ZSI framework. This way PyOPC can raise an OPC specific Python error, the *OPCServerError*. The OPCServerError inherits from ZSI's *Fault* error and contains the SOAP faultcode, the faultstring and the detail. More information about SOAP faults can be found in [SEE02] or [LIV02].

4 Client Functionality

The PyOPC framework enables access of OPC XML-DA compliant servers by providing classes that can be used to easily create OPC XML-DA clients.

PyOPC offers two different classes for this task:

- The *XDAClient* class that implements simple access of OPC servers. This class does not offer concurrent connections.
- PyOPC also provides the more complicated *TWXDAClient* class, which is based on the Twisted framework. *TWXDAClient* enables concurrent client connections by utilizing Twisted's event mechanism.

These classes are contained in two different Python modules, namely *XDAClient* and *TWXDAClient*) that have to be imported before the client classes can be used. As already shown in Listing 3, global options can be defined during object creation, which then apply to all OPC operations that are handled by this client object.

Most of these options are OPC-specific and are described in [OPCXMLDA]. However, the following options are PyOPC-specific or are automatically handled by the client object:

OPCServerAddress: This option specifies the address of the OPC XML-DA server, such as *http://path/to/server*. The OPCServerAddress option is mandatory and can only be applied during client object creation.

ClientRequestHandle/ClientItemHandle: These options may help the OPC client and server to distinguish between different client requests. If these options are not specified, they will be automatically generated by PyOPC.

4.1 Building OPC XML-DA Clients with the PyOPC XDAClient class

Listing 7 shows example code of a PyOPC XDAClient-based client[12] that first retrieves the server status, browses the root item and reads an item:

```
from PyOPC.OPCContainers import *
from PyOPC.XDAClient import XDAClient

def print_options((ilist,options)):
    print ilist; print options; print

address='http://violin.qwer.tk:8000/'

xda = XDAClient(OPCServerAddress=address,
                ReturnErrorText=True)

print_options(xda.GetStatus())
print_options(xda.Browse())
print_options(xda.Read([ItemContainer(ItemName='simple_item',
                                      MaxAge=500)],
                       LocaleID='en-us'))
```

Listing 7: Sample client code based on the PyOPC XDAClient module

[12] This sample code can also be found in the file *samples/clients/simple.py* in the PyOPC distribution

Line 1 and 2 import the needed PyOPC modules. In Line 4 a simple function is defined that prints a list of ItemContainer objects (*ilist*) and the global options Python dictionary (*Options*).

In line 9, the client object is created: As global options, the server address is specified and *ReturnErrorText = True* denotes that the client requests verbose error descriptions.

Lines 12, 13 and 14 show the three different OPC operations. The return parameters of these operations are a list of ItemContainer objects and the global options (a Python dictionary), which are both handled by the function *print_options*.

4.2 Building OPC XML-DA Clients with the PyOPC TWXDAClient class

The *XDAClient* module has the disadvantage that operations can only be handled sequentially. When certain OPC operations take significantly longer than others, it is a better solution to execute the requests in parallel. The Twisted framework introduces an event-based mechanism that is utilized by the the *TWXDAClient*, so that concurrent server requests can be made[13]. However, this Twisted-based client class is more complex than its simpler alternative. More information about the Twisted framework and its underlying concepts can be found in [FET06].

Listing 8 implements the same functionality as listing 7 but executes the three OPC operations concurrently:

```
from PyOPC.OPCContainers import *
from PyOPC.TWXDAClient import TWXDAClient
from twisted.internet import reactor

OPERATIONS = 3

def print_options((ilist,options)):
    print ilist; print options; print
    global OPERATIONS
    OPERATIONS -= 1
    if OPERATIONS == 0:
        reactor.stop()

def handleError(failure):
    print "An Error occured"
    print failure.getTraceback()
    reactor.stop()

address='http://violin.qwer.tk:8000/'

xda = TWXDAClient(OPCServerAddress=address,
                  ReturnErrorText=True)

d = xda.twGetStatus()
d.addCallback(print_options)
d.addErrback(handleError)

d = xda.twBrowse()
d.addCallback(print_options)
```

[13]This event-based mechanism must not be confused with multi-threading.

```
30 d.addErrback(handleError)

32 d = xda.twRead([ItemContainer(ItemName='simple_item', MaxAge=500)],
                   LocaleID='en-us')
34 d.addCallback(print_options)
   d.addErrback(handleError)
36
   reactor.run()
```

Listing 8: Sample client code based on the PyOPC TWXDAClient module

In line 3, certain Twisted modules are imported. Line 24-26 show how an OPC operation is done in "Twisted style": first, the method *twGetStatus* is called that returns a deferred. Then two functions are attached to this deferred, namely a "callback" method, which prints the results of the OPC operation and an "errback" method, which is executed when an error (failure) is returned.

In line 37, all deferreds are initialized, therefore the Twisted reactor is started, which triggers all deferreds. This way, all OPC operations are started and when the requested data is returned, the appropriate callback/errback methods are called.

As the execution order of the attached callback methods cannot be predicted, a global variable *OPERATIONS* is defined, which is used by the function *print_options* to stop the Twisted reactor when all pending server requests have finished.

5 Server Functionality

The PyOPC framework also enables easy and rapid creation of OPC XML-DA compliant servers. Implementing OPC servers is more complicated than creating clients, however PyOPC introduces several concepts that should greatly reduce the effort.

Most often, an OPC server will retrieve data from underlying devices or networks, such as fieldbuses. Most of these underlying technologies will also provide operations similar to OPC operations, such as reading and writing. In such a situation, the OPC XML-DA server will be similar to a proxy, which retrieves data on one side from fieldbuses or devices, reformats it and provides it to clients on the other side, such as depicted in figure 5.

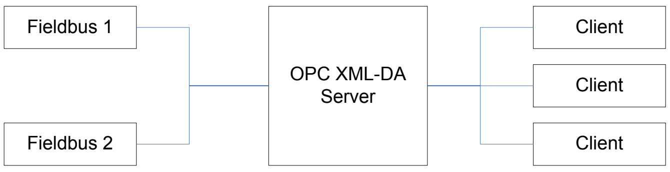

Figure 5: OPC XML-DA Server as a Proxy

PyOPC introduces a server class, the *XDAServer*, which provides methods for each OPC operation. This class can be inherited and the methods can be overridden by custom implementations, as illustrated in figure 6.

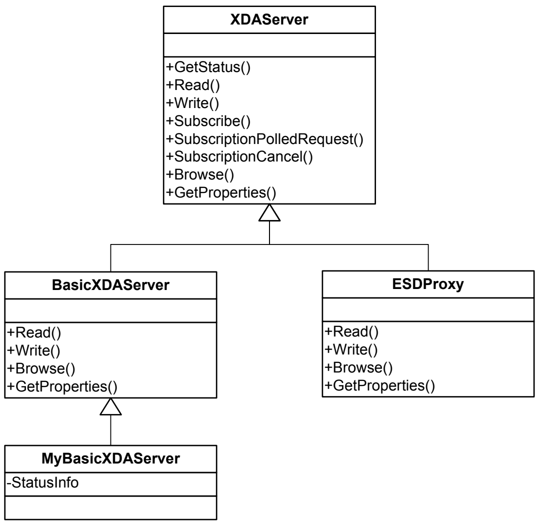

Figure 6: Server Class Hierarchy

For instance, the class *BasicXDAServer* inherits from *XDAServer*. The BasicXDAServer class overrides certain functionality of its parent class, such as Read and Write, where it implements its own functionality. However, this is still a quite general class and is not intended for the actual server instance. Instead, another class, *MyBasicXDAServer* inherits from the class, which defines various attributes that define the runtime parameters for the server instance, such as “GetStatus”, which may be set to any custom value.

This way, implementing a OPC XML-DA server with PyOPC leads to a three-level class hierarchy:

1. The basic XDAServer class that implements general functionality, which is provided by the PyOPC framework

2. A server-specific class that overrides certain operations to implement its custom functionality, which has to be implemented for the dedicated system

3. The production-specific class, which defines several production specific parameters and is used for the server instance.

Listing 9 shows the code for a very basic server that inherits from the BasicXDAServer class, which enables to define several OPC items in the server instance itself. These OPC items can then be read and written by OPC clients.

```
1  import random
   from twisted.internet import reactor,defer
3  from PyOPC.servers.basic import BasicXDAServer

5  # Read sample OPC items for testing
   import sample_items
7
   class MyXDAServer(BasicXDAServer):
9      OPCItems = sample_items.TestOPCItems
       StatusInfo = 'My Basic OPC XML-DA Server'
11
       def GetStatus(self, (IPH,inOptions,outOptions)):
13         ''' Custom GetStatus that alters the Product Version'''

15         outOptions['ProductVersion'] = str(random.choice(range(1,10)))

17         return super(MyXDAServer, self).GetStatus((IPH,inOptions,
               outOptions))
```

Listing 9: Simple OPC XML-DA Server

In line 9 some predefined OPC items are set in the server instance. These items are defined in an external module, which is imported in line 6. Moreover the MyXDAServer class defines the class attribute “StatusInfo” which will then be used in the OPC “GetStatus” operation.

Line 12-18 shows how the “GetStatus” operation is overridden by the inherited class. In this method, the OPC option “Product Version” is set to an arbitrary number between 1 and 9.

Line 17 is very important: this line calls the method of the parent class and returns the results. Python does not automatically call its parent class, this has to be done manually. However, it is mandatory in PyOPC that an overridden OPC operation has to call the method in its parent class. The reason is that this parent method fulfills several needed functionality, such as setting several other needed OPC options to maintain compatibility with the OPC XML-DA specification. The call of the superclass method has always to be done at the end of the custom method.

The above listing also shows how data such as the global options and the OPC items are passed. It is obvious that any XDAServer-based method, which represents an OPC operation, has to process parameters from the client request message and has to create several appropriate output parameters, which form the base of the response message. In PyOPC, these request and response parameters are passed from one method to another and thus also from each child to its parent method, such as shown in figure 7.

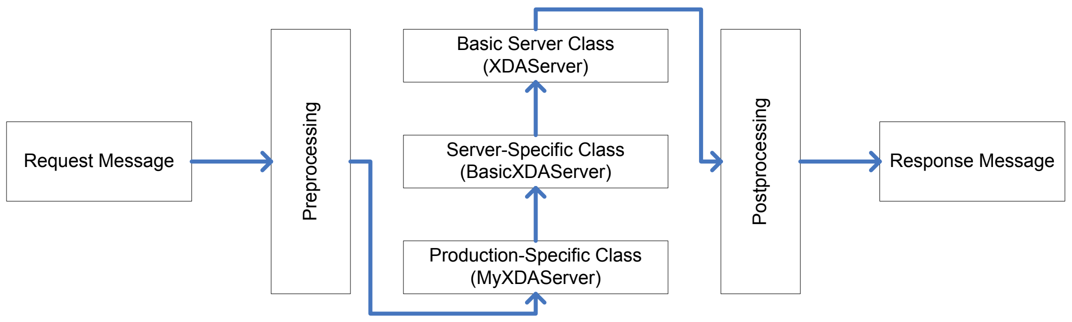

Figure 7: OPC Parameter Passing

Similar to the PyOPC-based client, the request and response messages are represented by a Python dictionary that contains the global OPC options and a list of ItemContainer objects, representing the OPC items. It can be observed in figure 7 that OPC data is passed from one method to another. All these methods require the input parameters and will set or alter certain output parameters.

Therefore it is appropriate to aggregate the input and output parameters in one Python object, which is then passed from one method to another. Therefore each PyOPC method, which represents an OPC operation, must define a Python tuple as a parameter which contains the following three objects:

1. The **Item Pair Holder** (IPH), a special object that contains an input and output list of ItemContainer objects. These lists have always to be of the same length and every input ItemContainer object has a corresponding output ItemContainer object. These internal ItemContainer lists are normally not directly accessed. Instead, the ItemPairHolder object implements an "append" method, which can be used to append an input and output ItemContainer object. Moreover the object is iterable, which is most often needed by OPC operation methods.

   Listing 10 shows how such an ItemPairHolder can be created and managed:

```
IPH = ItemPairHolder()
IPH.append(inItem, outItem)
for inItem, outItem in IPH:
    outItem.ItemPath = inItem.ItemPath
```

   Listing 10: Creating and Managing the ItemPairHolder object

   The example listing first creates an ItemPairHolder object and appends some predefined ItemContainer objects. Then one certain attribute of the output item, the "ItemPath" is set to its corresponding input attribute.

2. The **input options** (inOptions), a Python dictionary, which contains all global input options

3. The **output options** (outOptions), a Python dictionary, which contains all global output options

Server Configuration

The server can be configured through several class attributes. These attributes can either be overridden in the inheriting object, or can also be specified at the server object instantiation.

The basic *XDAServer* class provides the following options. (Their default values are given in parentheses.) Some of them are are further described in later sections:

AutoItemCache *(True)***:** This option enables the automatic OPC item caching.

WritePurgeCache *(True)***:** Denotes if the item cache should be flushed after an item is written.

DefaultMaxAge *(1000)***:** The default maximum age of an item.

BufferSize *(100)***:** The default subscription buffer size in number of items.

ThreadedParsing *(True)***:** One of the most CPU-intensive tasks is the parsing of the incoming SOAP message. In order to speed up the server, it is possible to execute the parser in a separate thread, so that the server can execute other requests in parallel[14].

ThreadPoolSize *(5)***:** The maximum amount of concurrent threads.

SubscriptionPingRate *(10000)***:** The default subscription ping rate in milliseconds.

MaxPingRate *(86400000 = 1 Day)***:** The maximum ping rate that may be specified by a client.

MaxSamplingRage *(100)***:** The maximum sampling rate in milliseconds that clients may specify.

HandleProperty_value/quality/timestamp/scanRate *(True)***:** Denotes if the above properties are automatically generated/handled.

5.1 Preprocessing and Postprocessing

The PyOPC XDAServer class automatically parses incoming SOAP messages and creates appropriate PyOPC objects, which represent the incoming message. After that, the preprocessing stage prepares and handles some of the following options of the outgoing message:

RcvTime: This option is set to the time, the SOAP message was received by the server.

ClientRequestHandle: The content of the incoming ClientRequestHandle is automatically copied to the outgoing message.

RevisedLocaleID: If the requested locale is not available, the server automatically chooses the first available locale and returns it with this options.

ServerState: This option is set to the ServerState attribute of the server class, the default is "running".

After this stage, the server operation methods are executed. Therefore the data from the preprocessing stage is available in these methods and may be modified. For instance, the read operation may check for the option "RevisedLocaleID" in the outgoing message and set it to a different locale.

After the server operations are finished, the PyOPC objects, which represent the incoming and outgoing SOAP messages is handed over to the postprocessing stage, which handles and modifies the following options:

[14]The speed gain is currently not very significant, as fast XML parsers are quite efficient compared to other tasks, such as serializing the SOAP message.

Unhandled Items: As denoted above, the incoming and outgoing OPC items are stored in the ItemPairHolder object. The preprocessing stage will create this object, which contains an incoming ItemContainer object and an associated, empty outgoing ItemContainer object. The operations should then fill this outgoing item with appropriate data. If, however, the outgoing ItemContainer object is still empty[15] in the postprocessing stage, an error of the type "PYO_E_EMPTYITEM" is associated with it, denoting that the item contains no data.

ErrorText: If *ReturnErrorText* is set to false in the request message, any error text in the outgoing message is deleted. Otherwise the error text will be kept, and if there is none in the outgoing message, it will be set to a blank string.

DiagnosticInfo: If *ReturnDiagnosticInfo* is set to false or is omitted in the request message, any diagnostic info in the outgoing message is deleted. Otherwise the diagnostic info will be kept, and if there is none in the outgoing message, it will be set to a blank string.

Timestamp: If *ReturnItemTime* is set to false or is omitted in the request message, any item-related timestamp in the outgoing message is deleted. Otherwise the timestamp will be kept, and if there is none in the outgoing message, it will be set to the current time.

ItemPath/ItemName: If *ReturnItemPath/Name* is set to false or is omitted in the request message, any ItemPath/Name in the outgoing message is deleted. Otherwise the ItemPath/Name will be kept, and if there is none in the outgoing message, it will be set to a blank string.

ClientItemHandle: The ClientItemHandle of each incoming item is copied to the according outgoing item.

ReplyTime: The ReplyTime in the outgoing message is set to the current time, unless it has not been already set a previous server method.

5.2 Item Caching and Subscriptions

PyOPC provides support for advanced OPC XML-DA services, such as subscriptions and Item Caching. Unless these services are disabled, they are automatically available at any PyOPC-based OPC server.

Item Caching

OPC XML-DA servers normally retrieve data from underlying systems, such as fieldbuses. Different client will then access these items through the OPC server. There may be situations, where many clients access the same OPC item over and over. These client requests will therefore lead to a significant load on underlying systems and may even exceed their capabilities.

One solution to this problem is caching: when the OPC server retrieves an item from an underlying system, it stores it for a predefined amount of time. These cached items are then

[15]A ContainerItem is empty, if the attribute 'IsEmpty' is set to true. This happens if it is created such as i=ItemContainer() and no attribute is ever set.

available for OPC clients, therefore client requests do not necessarily lead to data retrieval from underlying systems.

In the client request message, the option "MaxAge" may be specified, denoting how old the item may be. If MaxAge is greater than the time the item is cached, the server will build the response message upon the cached item, otherwise the server will retrieve new data.

The PyOPC framework does automatically implement OPC item caching, if the server attribute "AutoItemCache" is set to true (which it is by default). The developer can also specify the attribute "DefaultMaxAge", which defines the maximum age in milliseconds for requests that do not provide the MaxAge option.

The exact item caching mechanism is illustrated in figure 8:

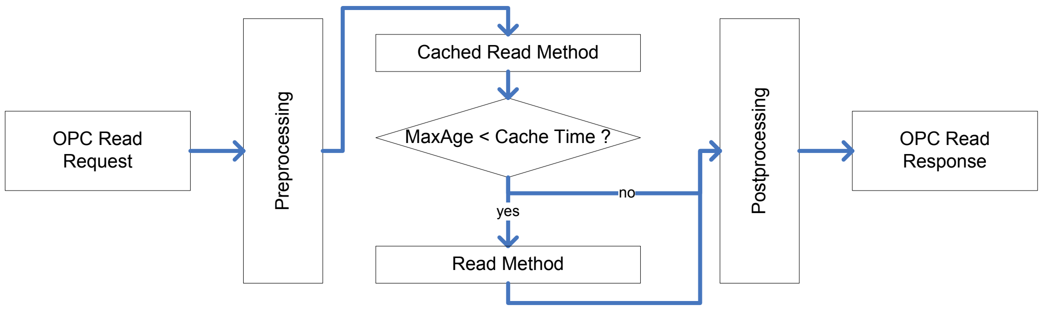

Figure 8: Mechanism of PyOPC Item Caching

If an OPC Read request message is received by the server and "AutoItemCache" is set to true, the XDAServer method "CachedRead" is called. This method decides whether to return cached data regarding the MaxAge parameter or whether it calls the server's read method, which retrieves the data from the underlying system.

This way, the developer does not need to regard item caching, he has to code only the data retrieval itself. His read method will be only called if the maximum allowed age is smaller than the time, the item is cached.

Write operations update data in underlying devices. Therefore by default the read cache of a written item is flushed. However, in certain situations it may be appropriate not to clear the cache. Therefore PyOPC provides the option "WritePurgeCache" which can be set to false to omit the automatic cache flushing.

Subscriptions

In order to observe OPC items, one could theoretically periodically poll OPC items with read operations and check for changes. However, such polling leads to an unnecessary network and server load. Therefore [OPCXMLDA] introduces so-called "subscriptions" and specifies three related OPC operations.

OPC clients may subscribe to items, which basically commands the server to observe these items for changes. Such changes will be stored by the server and can later be retrieved by the client.

Implementing such a subscription mechanism in the server is quite complicated. The OPC XML-DA specification describes several complex issues, such as deadband (recording items only if they exceeded a predefined value) and the so-called extended subscription architecture. A detailed description of these topics is given in [OPCXMLDA].

To ease the development of OPC XML-DA servers, the PyOPC framework implements a default mechanism for subscriptions which is automatically available and functional for all

OPC XML-DA servers based on the *XDAServer* class. The basic idea is that most underlying systems such as fieldbuses have a read operation that is roughly similar between different systems, while system notifications - a common way to observe datapoints for changes - can be very different on fieldbus systems. Therefore the most compatible way is to utilize the read operation for subscriptions.

Therefore developers only have to implement the read operation and will thus automatically enable subscriptions. The basic mechanism is illustrated in figure 9.

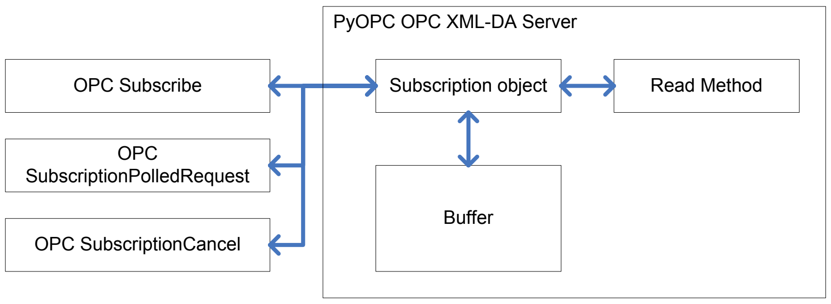

Figure 9: Subscriptions in PyOPC

It can be seen that a client can utilize the following three operations to handle subscriptions:

- *Subscribe* creates a subscription object in the server. This subscription object will then utilize the server's read operation to periodically poll the underlying system for changed items. These changed items are then stored - either in the subscription object itself, or, if specified by the client, in a subscription buffer, which is shared among all subscriptions.
- *SubscriptionPolledRequest (SPR)* is then used by clients to retrieve all changed values.
- *SubscriptionCancel* can then be used to cancel the subscription which results in the deletion of the subscription object.

PyOPC also covers all complex subscription-related issues, such as the PingRate, HoldTime/WaitTime, deadband and more, therefore developers do not have to deal with these issues.

The following functionality can be configured in the server object:

BufferSize: All OPC subscriptions share one buffer, which is used to store changed items until they are fetched by the client via the SPR operation. With the BufferSize attribute, the number of items can be specified that the buffer can hold. If more items than this number are stored in the buffer, the oldest entries are lost.

SubscriptionPingRate: If the client did not specify a ping rate, this predefined value will be used. The ping rate defines, how much time may elapse between two client polls. If the ping rate is exceeded, the subscription is automatically canceled.

MaxPingRate: This is the maximum ping rate, a client may specify.

MaxSamplingRate: As denoted above, the subscription object periodically fetches (samples) item values from underlying devices. This sampling can be very demanding for the underlying system, therefore a maximum sampling rate can be specified in the server. If a client requests a higher sampling rate than this value, it is revised to this value by the server.

OPC server developers should set this to a suitable value: the maximum sampling rate should never be higher than the time that is needed for a read request.

It should be denoted that PyOPC allows only one SPR for one subscription object at a time. If a second concurrent SPR is issued, the server will return an error[16].

An OPC server developer may also implement his own subscription mechanisms. This can be done by overriding the three XDAServer methods *Subscribe*, *SubscriptionPolledRequest* and *SubscriptionCancel*, however, this will most often not be necessary.

5.3 Operation Specific Functionality and Other Issues

Automatic Readback

In OPC write requests, the client may specify the option "ReturnValuesOnReply". If this option is set to true, the server will read back the written values. This is accomplished by calling the servers read method.

Browsing

Clients may specify OPC properties in a browse request, which will then be returned along with the appropriate browse result. In order to retrieve these properties, the server's Browse method will call the servers GetProperties method.

Property Handling

Certain OPC properties, namely *value, quality, timestamp* and *scanRate* can be automatically handled by the PyOPC framework. This way, these four properties will be automatically available for all OPC items.

The values of these properties will be retrieved as follows:

- The values of the *value, quality* and *timestamp* properties are simply retrieved with the servers read method, which contains all needed data.
- The value of *scanRate* is set to MaxSamplingRate.

Logging

PyOPC based servers also support detailed logging and provides the following three different logs:

Access Logging: If a client accesses the PyOPC server, the clients IP address and the requested SOAPAction, which defines the OPC operation is logged in this file. The default file name for the access log is "access.log".

Error Logging: PyOPC server errors are kept in this log. Its default name is "error.log".

[16]The OPC XML-DA specification is quite loose on this topic, therefore in PyOPC concurrent SPRs are prohibited.

Debug Information: During development and tests, it is often interesting for the programmer to have access to further information, especially to the client/server communication. Therefore PyOPC logs the SOAP messages along with the HTTP header in a pretty-printed style. By default the debug log is disabled.

The file names of these logs can be configure by setting the server attributes “access_log_fn”, “error_log_fn” and “http_log_fn” to the desired name. Moreover, logging can also be omitted by setting one or more of these attributes to a blank string.

Setting Up an OPC Server Instance

As the server classes of PyOPC are based on the Twisted framework, it is also utilized to set up the OPC server instance. This can be done as shown in listing 11, which implies the proper definition of the class “MyXDAServer”, as already shown in listing 9.

```
  from twisted.web import resource, server
2 xdasrv = MyXDAServer(http_log_fn = 'http.log')
  root = resource.Resource()
4 root.putChild('',xdasrv)
  site = server.Site(root)
6 reactor.listenTCP(8000, site)
  reactor.run()
```

Listing 11: Instantiating and Starting a PyOPC-based OPC XML-DA Server

Line 1-2 import all needed Twisted modules and create the PyOPC server object. In line 3, a Twisted resource object is created, where servers can be added, such as shown in line 4. Line 5-7 then starts the server. This server will then be reachable under the address “http://server:8000/”.

A Twisted resource.Resource object is not limited to one PyOPC server instance, instead it is possible to add multiple server objects, such as shown in listing 12, which are then reachable under different URLs.

```
root = resource.Resource()
root.putChild('srv1',MyXDAServer1())
root.putChild('srv2',MyXDAServer2())
root.putChild('srv3',MyXDAServer3())
```

Listing 12: Adding Multiple PyOPC Server Objects to One Twisted Resource

5.4 Contributed Servers

The PyOPC framework implements two simple servers, which on the one hand can be seen as a reference design and may on the other hand be used for setting up simple test servers.

BasicXDAServer

This simple server does not retrieve data from other resources, instead the OPC item data can be directly defined in the server.

Listing 13 shows the code of a server that is based on the BasicXDAServer class:

```
class MyXDAServer(BasicXDAServer):
    OPCItems = (ItemContainer(ItemName='sample_integer',
                              Value=14,
                              QualityField='good'),
                ItemContainer(ItemName='sample_float',
                              Value=96.43,
                              QualityField='good'))
```

Listing 13: PyOPC server based on the class BasicXDAServer

This server defines two OPC items, which can then be accessed by OPC XML-DA clients.

ESDProxy

This server can be seen as a reference design for OPC servers that retrieve OPC data from external sources. A detailed description of this server can be found in [HIM06].

s

A Appendix A - Local/Global OPC Options

The OPC XML-DA specification defines various options, which can be global and/or local and are associated with one or more OPC operations. A detailed description of these options can be found in [OPCXMLDA]. However, the specification is somehow complicated and some of these options can not be easily found. Moreover these options will be represented by specific Python data types.

In order to ease the client/server development with PyOPC, table 2 and 3 outline what options are used by which functions. The letters "G" and "L" indicate if the option is global or local for an OPC operation.

Moreover an alphabetical reference of all available OPC options is given in this appendix, which outlines the functionality of an OPC option and moreover denotes, by which Python data type it is represented in the PyOPC framework.

BrowseFilter (*string*) Limits the returned elements during a browse operation. Allowed values are *all, branch, item.*

ClientItemHandle (*string*): An identifier of the item in a request message. The ClientItemHandle is returned along with the requested item.

ClientRequestHandle (*string*): An identifier of the client request.

ContinuationPoint (*string*): A browse option for specifying secondary browse requests.

DataBufferOverflow (*bool*): Indicates if some item changes were lost during subsequent subscription polls

Deadband (*float*): The percentage of an item value change which has to be exceeded so that the item "has changed", meaning that the item will be sent during the next subscription poll.

DiagnosticInfo (*string*): Additional server specific information in case of an error.

Description (*string*): A verbose description of an item property

ElementNameFilter (*string*): Limits the returned elements during a browse operation

EnableBuffering (*bool*): Denotes if changed values should be buffered in case of a subscription

ErrorText (*string*): Verbose error description in case of an item error

HasChildren (*bool*): Indicates if a browse element has child elements

HoldTime (*datetime*): Time to wait until the subscription poll response is sent

InvalidServerSubHandles (*list of strings*): A list of invalid ServerSubHandles

IsItem (*bool*): Indicates if a browse element is an item

ItemName (*string*): The ItemName is part of the namespace of an OPC item. Together with the ItemPath, it forms a unique identification of an item.

	GetStatus	GetStatus Response	Read	Read Response	Write	Write Response	Subscribe	Subscribe Response	SubscriptionPolledRequest	SubscriptionPolledRequest Response	SubscriptionCancel	SubscriptionCancel Response	Browse	Browse Response	GetProperties	GetProperties Response
BrowseFilter													G			
ClientItemHandle			L	L	L	L	L	L		L						
ClientRequestHandle	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G
ContinuationPoint													G	G		
DataBufferOverflow										G						
Deadband							G/L									
DiagnosticInfo				L	L	L		L		L						
Description														L		L
ElementNameFilter													G			
EnableBuffering							G/L									
ErrorText				L		L		L		L				L		L
HasChildren														L		
HoldTime									G							
InvalidServerSubHandles										G						
IsItem														L		
ItemName			L	L	L	L	L	L		L			G	L	L	L
ItemPath			G/L	L	G/L	L	G/L	L		L			G	L	G/L	L
LimitField				L	L	L		L		L						
LocaleID	G		G		G		G		G				G		G	
MaxAge			G/L													
MaxElementsReturned													G			
MoreElements														G		
Name														L		L
ProductVersion		G														
PropertyNames													G		G	
QualityField				L	L	L		L		L						
RcvTime		G		G		G		G		G				G		G
ReplyTime		G		G		G		G		G				G		G

Table 2: OPC Options and Operations

ItemPath (*string*): The ItemPath is part of the namespace of an OPC item. Together with the ItemName, it forms a unique identification of an item.

LimitField (*string*): Transports the limit status of an OPC item and may be one of the following string: *none*, *low*, *high*, *constant*.

LocaleID (*string*): Requested locale for the return message.

MaxAge (*long*): Denotes how old the item value may be in milliseconds. MaxAge can be utilized for item caching in the server.

	GetStatus	GetStatus Response	Read	Read Response	Write	Write Response	Subscribe	Subscribe Response	SubscriptionPolledRequest	SubscriptionPolledRequest Response	SubscriptionCancel	SubscriptionCancel Response	Browse	Browse Response	GetProperties	GetProperties Response
RequestDeadline			G		G		G		G							
RequestedSamplingRate							G/L									
ReqType			G/L				G/L									
ResultID				L	L	L		L		L				L		L
ReturnAllItems									G							
ReturnAllProperties													G		G	
ReturnDiagnosticInfo			G		G		G		G							
ReturnErrorText			G		G		G		G				G		G	
ReturnItemName			G		G		G		G							
ReturnItemPath			G		G		G		G							
ReturnItemTime			G		G		G		G							
ReturnPropertyValues													G		G	
ReturnValuesOnReply					G		G									
RevisedLocaleID		G		G		G		G		G				G		G
RevisedSamplingRate								G/L								
ServerState		G		G		G		G		G				G		G
ServerSubHandle								G			G					
ServerSubHandles									G							
StartTime		G														
StatusInfo		G														
SubscriptionPingRate								G								
SupportedLocaleIDs		G														
SupportedInterfaceVersions		G														
Timestamp				L	L	L		L		L						
Value				L	L	L		L		L				L		L
ValueTypeQualifer				L	L	L		L		L						
VendorField				L	L	L		L		L						
VendorFilter													G			
VendorInfo		G														
WaitTime									G							

Table 3: OPC Options and Operations

MaxElementsReturned (*long*): Maximum amount of returned elements during a browse request.

MoreElements (*bool*): Indicates that there are more elements than the returned ones in a browse response message

Name (*string/QName*): This option is used as an identifier for a browse element (string type) or a property (QName type).

ProductVersion (*string*): A version string of the OPC server

PropertyNames (*list of QNames*): A list of item properties that should be returned.

QualityField (*string*): The quality of an OPC item value. [OPCXMLDA] specifies a predefined list of allowed qualities, such as “good”, “uncertain” or “bad”.

RcvTime (*datetime*): The time the server received the request.

ReplyTime (*datetime*): The time the server returned the response.

RequestDeadline (*datetime*): The time until the server response has to be issued.

RequestedSamplingRate (*long*): The time in milliseconds in which the server checks for value changes in case of a subscription.

ReqType (*QName*): With this options, the client may specify the data type of an OPC item value. The available data types are listed in [OPCXMLDA].

ResultID (*QName*): In case of an item error, this option specifies the error type.

ReturnAllItems (*bool*): Indicates if the server should return only the changed or all OPC items during a subscription poll

ReturnAllProperties (*bool*): Indicates if all item properties should be returned

ReturnDiagnosticInfo (*bool*): Return additional server specific information in case of an error

ReturnErrorText (*bool*): If True, the OPC server returns a verbose error description in case of an item error.

ReturnItemName (*bool*): Indicates whether the ItemName is returned by the server

ReturnItemPath (*bool*): Indicates whether the ItemPath is returned by the server

ReturnItemTime (*bool*): Indicates whether the Timestamp of an OPC item is returned by the server

ReturnPropertyValues (*bool*): Indicates whether all item property values should be included in a response message

ReturnValuesOnReply (*bool*): Specifies if the item values should be included in the response message

RevisedLocaleID (*string*): In case the requested locale is not implemented by the server, it is revised. The revised locale is sent back to the client by the RevisedLocaleID option.

RevisedSamplingRate (*long*): If the OPC server does not the requested subscription sampling rate, it returns a revised rate in this option.

ServerState (*string*): The current status of the OPC server, the values may be one of the following: *running, failed, noConfig, suspended, test, commFault*

ServerSubHandle (*string*): An identifier of an OPC subscription

ServerSubHandles (*list of strings*): A list of ServerSubHandles.

StartTime (*datetime*): The time the OPC server was started

StatusInfo (*string*): Provides additional server information.

SubscriptionPingRate (*long*): Maximum time in milliseconds between SubscriptionPolledRequest operations. If the ping rate is exceeded, the subscription will be canceled.

SupportedLocaleIDs (*string*): String that contains all supported locales by the server

SupportedInterfaceVersions (*string*): Supported versions of the OPC XML-DA standard, currently only "XML_DA_Version_1_0" is allowed

Timestamp (*datetime*): The time when the OPC item value was sampled

Value (*anyType*): The value of an OPC item or OPC property

ValueTypeQualifier (*QName*): In case the value is date/time based, it will identifiy the exact XML-Schema data type.

VendorField (*long*): A numeric value that matches the OPC Vendor Bit Field

VendorFilter (*string*): Limits the returned elements during a browse operation

VendorInfo (*string*): Vendor specific server information

WaitTime (*long*): Time in milliseconds which the server should wait during a pending subscription poll for item value changes

References

[BIR01] **Mark Birbeck, Jason Diamond, Jon Ducket et al.**:
Professional XML 2nd Edition
Wrox Press Ltd. **2001**

[FET06] **Abe Fettig**:
Twisted Network Programming Essentials O'Reilly, CA-95472 **2006**

[HIM06] **Hermann Himmelbauer**:
SOAP Interface For An Internet/Fieldbus Gateway
Vienna University of Technology, Austria **2006**

[HUN03] **Andy Hunt / Dave Thomas**:
Pragmatic Unit Testing in Java with JUnit The Pragmatic Programmers, LLC **2003**

[LIV02] **Dan Livingston**:
Advanced SOAP Web Development
Prentice-Hall Inc., NJ-07458 **2002**

[MAR03] **Alex Martelli**:
Python in a Nutshell
O'Reilly, CA-95472 **2003**

[OPCDA] **OPC Foundation**:
OPC Data Access Custom Interface Standard Version 3.00
OPC Foundation, AZ-85260-1830, **2003**

[OPCXMLDA] **OPC Foundation**:
OPC XML-DA Specification Version 1.01
OPC Foundation, AZ-85260-1830, **2004**

[PIL05] **Mark Pilgrim**:
Dive Into Python
Apress CA-94710 **2005**

[SEE02] **Scott Seely**:
SOAP: Cross Platform Web Service Development using XML
Prentice-Hall Inc., NJ-07458 **2002**

[VLI02] **Eric van der Vlist**:
XML Schema
O'Reilly, CA-95472 **2002**

[WAL02] **Aaron E. Walsh**:
UDDI, SOAP and WSDL - The Web Services Specification Reference Book
Prentice-Hall Inc., NJ-07458 **2002**