

---

# **pyrfc Documentation**

***Release 1.9.4***

**SAP**

April 01, 2016



<b>1</b>	<b>Documentation</b>	<b>3</b>
1.1	Introduction . . . . .	3
1.2	Installation . . . . .	4
1.3	Client scenario . . . . .	9
1.4	Server scenario . . . . .	14
1.5	Security . . . . .	20
1.6	Building from source . . . . .	24
1.7	Remarks . . . . .	26
1.8	Bibliography . . . . .	27
<b>2</b>	<b>API documentation</b>	<b>29</b>
2.1	pyrfc . . . . .	29
<b>3</b>	<b>Change log</b>	<b>33</b>
3.1	Change log . . . . .	33
<b>4</b>	<b>Indices and tables</b>	<b>37</b>



The [pyrfc Python package](#) provides Python bindings for *SAP NetWeaver RFC Library*, for a comfortable way of calling ABAP modules from Python and Python modules from ABAP, via SAP Remote Function Call (RFC) protocol.

It was inspired by [Piers Harding's sapnwrfc package](#), wrapping the existing *SAP NetWeaver RFC Library* and rewritten using *Cython*.

To start using `pyrfc` follow the [Installation](#) guide.



---

## Documentation

---

### 1.1 Introduction

The Python connector (a synonym for the `pyrfc` package) wraps the existing *SAP NW RFC Library*, often colloquially called *SAP C connector* or *SAP NW RFC SDK*. To start using `pyrfc` and similar connectors effectively, we highly recommend reading a series of insightful articles about RFC communication and *SAP NW RFC Library*, published in the SAP Professional Journal (SPJ), in 2009, by Ulrich Schmidt and Guangwei Li: *Improve communication between your C/C++ applications and SAP systems with SAP NetWeaver RFC SDK* [Part 1: RFC Client Programming](#), [Part 2: RFC Server Programming](#), [Part 3: Advanced Topics](#).

The lecture of these articles and [NW RFC SDK Guide \(SAP Help\)](#) are recommended as an introduction into RFC communication and programming, while `pyrfc` documentation is focused merely on technical aspects of `pyrfc` API.

#### 1.1.1 Example usage

In order to call remote enabled ABAP function module, we need to open a connection with valid logon credentials.

```
from pyrfc import Connection
conn = Connection(user='me', passwd='secret', ashost='10.0.0.1', sysnr='00', client='100')
```

Using an open connection we can call remote enabled ABAP function modules from Python.

```
result = conn.call('STFC_CONNECTION', REQUTEXT=u'Hello SAP!')
print result
{u'ECHOTEXT': u'Hello SAP!',
 u'RESPTEXT': u'SAP R/3 Rel. 702   Sysid: ABC   Date: 20121001   Time: 134524   Logon_Data: 100/M
```

Finally, the connection is closed automatically when the instance is deleted by the garbage collector. As this may take some time, we may either call the `close()` method explicitly or use the connection as a context manager:

```
with Connection(user='me', ...) as conn:
    conn.call(...)
# connection automatically closed here
```

#### 1.1.2 Functional coverage

The goal of the Python connector development was to provide a package for interacting with SAP ABAP systems on an intuitive and adequate abstract level. Not each and every available function provided by *SAP NW RFC Library* is therefore wrapped into Python, but classes and methods are implemented, covering the most of the use cases relevant for projects done so far. The drawback of this approach is that fine-grained RFC manipulation is not possible sometimes but coverage can be extended if needed.

In line with this approach, we distinguish between two basic scenarios:

- Client, Python client calls ABAP server
- Server, ABAP client calls Python server

The coverage is as follows:

	Client	Server
Standard functionality, e.g. invoking arbitrary RFC	yes	yes (1)
Transactions (tRFC/qRFC)	yes	no
Background RFC	yes (2)	no
RFC Callbacks	no	no
Secure network connect (SNC)	yes	yes (1)
Single Sign on (SSO)	no	no

---

**Note:**

1. Server functionality is currently not working with Python 32bit under Windows.
  2. Background RFC is currently not working.
  3. In projects done so far mainly the Client scenario is used and lot more tested then Server. The Server scenario is therefore considered experimental for now.
- 

## 1.2 Installation

Python connector is a wrapper for the *SAP NetWeaver RFC Library* and you need to obtain and install it first.

If Python is not already installed on your system, you need to download and install Python as well.

After having *SAP NW RFC Library* and Python installed on your system, you can download and install one of provided `pyrfc` eggs, relevant for your platform and start using `pyrfc`.

You can also clone this repository and build `pyrfc` from the source code, following *Building from source*.

### 1.2.1 SAP NW RFC Library Installation

The entry page for *SAP NetWeaver RFC library* is SAP Service Marketplace (SMP), <http://service.sap.com/rfc-library>, with detailed instructions how to [download](#), [use](#) and [compile](#).

Basically, you should search for SAP NW RFC SDK 7.20, in Software Downloads of SAP Software Download Center on [SMP Support Portal](#), download SAP NW RFC Library adequate for your platform and Python version combination (see the matrix below) and unpack using SAPCAR archive utility.

SAPCAR can be downloaded from the SMP as well and you should search for SAPCAR 7.20 Which SAP NW RFC Library version is relevant for your platform? Here are platform/Python combinations tested so far:

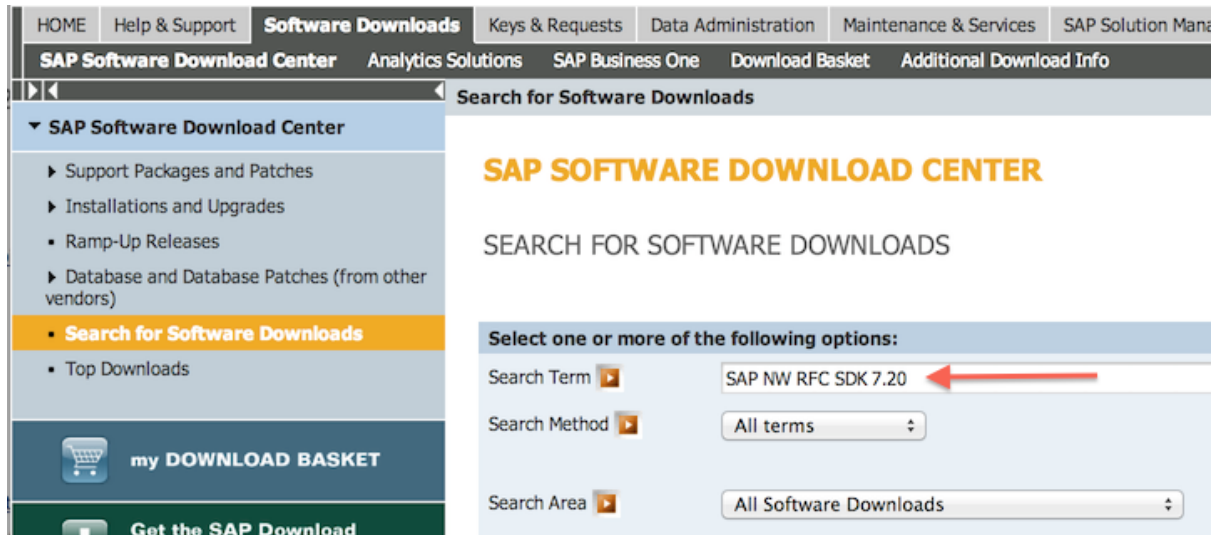
Platform	Python version	NetWeaver RFC Library (SMP)	Filename (SMP)	Python egg
Windows	Python 2.7 (32bit)	<i>Windows Server on IA32 32bit</i>	NWRFC_20-20004566	<code>pyrfc-1.9.3-py2.7-win32.egg</code>
Windows	Python 2.7 (64bit)	<i>Windows on x64 64bit</i>	NWRFC_20-20004568	<code>pyrfc-1.9.3-py2.7-win-amd64.egg</code>
Linux	Python 2.7 (64bit)	<i>Linux on x86_64 64bit</i>	NWRFC_20-20004565	<code>pyrfc-1.9.3-py2.7-linux-x86_64.egg</code>

---

**Note:**

- *SAP NW RFC Library* is fully backwards compatible and it is recommended using the newest version also for older backend system releases





HOME Help & Support **Software Downloads** Keys & Requests Data Administration Maintenance & Services SAP Solution Man



**SAP Software Download Center** Analytics Solutions SAP Business One Download Basket Additional Download Info



Search for Software Downloads



**SAP SOFTWARE DOWNLOAD CENTER**


SEARCH FOR SOFTWARE DOWNLOADS


Select one or more of the following options:

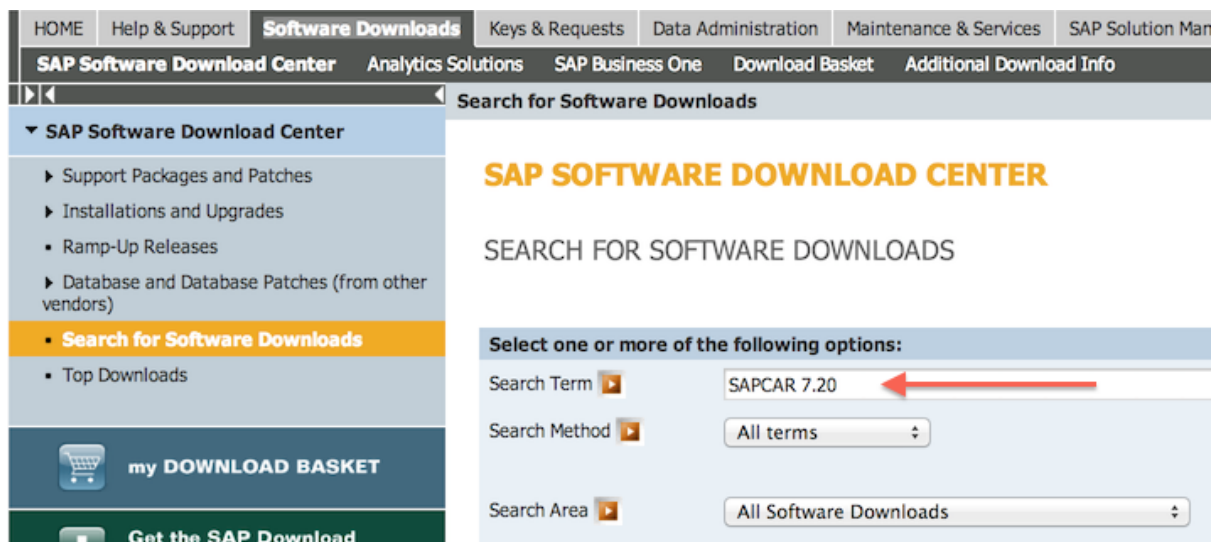
Search Term  SAP NW RFC SDK 7.20 

Search Method  All terms 

Search Area  All Software Downloads 

 my DOWNLOAD BASKET

 Get the SAP Download



HOME Help & Support **Software Downloads** Keys & Requests Data Administration Maintenance & Services SAP Solution Man



**SAP Software Download Center** Analytics Solutions SAP Business One Download Basket Additional Download Info



Search for Software Downloads



**SAP SOFTWARE DOWNLOAD CENTER**


SEARCH FOR SOFTWARE DOWNLOADS


Select one or more of the following options:

Search Term  SAPCAR 7.20 

Search Method  All terms 

Search Area  All Software Downloads 

 my DOWNLOAD BASKET

 Get the SAP Download

- SMP search terms and filenames given here will not be regularly updated, you should always search for current version or filename in `Software Downloads`.
  - The server functionality is currently not working under Windows 32bit
- 

The Python connector relies on *SAP NW RFC Library* and must be able to find library files at runtime. Therefore, you might either install the *SAP NW RFC Library* in the standard library paths of your system or install it in any location and tell the Python connector where to look.

Here are configuration examples for Windows and Linux operating systems.

## Windows

1. Create an directory, e.g. `c:\nwrfdc SDK`.
2. Unpack the SAR archive to it, e.g. `c:\nwrfdc SDK\lib` shall exist.
3. Include the `lib` directory to the library search path on Windows, i.e. *extend* the `PATH` environment variable.

## Linux

1. Create the directory, e.g. `/usr/sap/nwrfdc SDK`.
2. Unpack the SAR archive to it, e.g. `/usr/sap/nwrfdc SDK/lib` shall exist.
3. Include the `lib` directory in the library search path:
  - As root, create a file `/etc/ld.so.conf.d/nwrfdc SDK.conf` and enter the following values:

```
# include nwrfdc SDK
/usr/sap/nwrfdc SDK/lib
```

- As root, run the command `ldconfig`.

## 1.2.2 Python Connector Installation

### Windows

- If not already installed, you need to install Python first.  
First, decide whether you want to go with the 32bit or 64bit version and use standard Windows installers  
Python 2.7 (32 bit), <http://www.python.org/ftp/python/2.7.6/python-2.7.6.msi>  
Python 2.7 (64 bit) <http://www.python.org/ftp/python/2.7.6/python-2.7.6.amd64.msi>  
Add Python and Scripts directories to `PATH` environment variable, e.g.

```
set PATH=c:\Python27;c:\Python27\Scripts;%PATH%
```

- Install `easy_install`  
Use the distribute implementation of `easy_install` by downloading [https://bootstrap.pypa.io/ez\\_setup.py](https://bootstrap.pypa.io/ez_setup.py) and running

```
python ez_setup.py
```

**Note:** At this point you may like to install the `pip` package which extends the functionality of `easy_install`. However, `pip` cannot handle binary build distributions, which will be used later.

If you are in a internal network that uses a proxy to access resources from the internet, you may encounter *connection problems*.

---

- Virtual environment (optional)

You may now create an *virtual environment* and activate it.

- Install the Python connector

Open the command prompt with administrator rights, change to the `pyrfc\dist` directory and install adequate `pyrfc` egg. You need administrator rights, otherwise `easy_install` will open a new window and close it after execution – leaving you without the option to see what was done or what was the error.

```
easy_install <egg name>
```

Please look up the correct *egg name* depending on your platform and Python version.

- Run `python` and type `from pyrfc import *`. If this finishes silently, without output, the installation was successful.

## Python on Linux

- Install Python 2.7 (64bit, usually the default) via your preferred package manager
- Install `easy_install`

Use the distribute implementation of `easy_install` by downloading [https://bootstrap.pypa.io/ez\\_setup.py](https://bootstrap.pypa.io/ez_setup.py) and running

```
python ez_setup.py
```

**Note:** At this point you may like to install the `pip` package which extends the functionality of `easy_install`. However, `pip` cannot handle binary build distributions, which will be used later.

If you are in a internal network that uses a proxy to access resources from the internet, you may encounter *connection problems*.

- Virtual environment (optional)

You may now create an *virtual environment* and activate it.

- Install the Python connector:

```
easy_install <egg name>
```

Please look up the correct *egg name* depending on your platform and Python version.

- Run `python`, type `from pyrfc import *` and if it finishes silently, without any output, the installation was successful.

### 1.2.3 Virtual environments

We recommend using a *virtual environment* for the installation. This allows you to isolate the Python connector installation from your system wide Python installation.

We will now show the example usage for a Windows user that wants to create a virtual environment in `C:\PythonVE\py27-pyrfc`.

1. Install `virtualenv` on your system.

```
C:\>pip virtualenv
```

2. Open a command prompt and change to a directory where you want to create a virtual environment and create a virtual environment.

```
C:\>cd PythonVE
C:\PythonVE\>virtualenv --distribute --no-site-packages py27-sapwnrfc2
```

(Since `virtualenv` version 1.7, the `--no-site-packages` option is the default and can be omitted.)

3. Activate the environment via

```
C:\PythonVE\>cd py27-pyrfc
C:\PythonVE\py27-pyrfc\>Scripts\activate.bat
(py27-pyrfc) C:\PythonVE\py27-pyrfc\>
```

(On Linux use `source bin/activate`.)

4. After working on your project, you leave the virtual environment with

```
(py27-pyrfc) C:\PythonVE\py27-pyrfc\>deactivate
C:\PythonVE\py27-pyrfc\>
```

## 1.2.4 Problems

### Behind a Proxy

If you are within an internal network that accesses the internet through an HTTP(S) proxy, some of the shell commands will fail with `urlopen` errors, etc.

Assuming that your HTTP(S) proxy could be accessed via `http://proxy:8080`, on Windows you can communicate this proxy to your shell via:

```
SET HTTP_PROXY=http://proxy:8080
SET HTTPS_PROXY=http://proxy:8080
```

or permanently set environment variables.

### SAP NW RFC Library Installation

1. `ImportError: DLL load failed: The specified module could not be found.`  
(Windows) This error indicates that the Python connector was not able to find the C connector on your system. Please check, if the `lib` directory of the C connector is in your `PATH` environment variable.
2. `ImportError: DLL load failed: %1 is not a valid Win32 application.`  
(Windows) This error occurs when SAP NW RFC Library 64bit version is installed on a system with 32bit version Python.

### Environment variables

#### Windows

The environment variable may be set within a command prompt via the `set` command, e.g.

- `set PATH=%PATH%;C:\nwrfsdk\lib` (extend `PATH` with the C connector lib)
- `set HTTPS_PROXY=proxy:8080` (setting an proxy for HTTPS communication)

When the command prompt is closed, the environment variable is reset. To achieve a persistent change of the environment variable, do the following (Windows 7):

1. Open the Start Menu and type `environment` into the search box.
2. A window opens in which the user variables are displayed in the upper part and the system variables in the lower part. You may select and edit the desired variable.
3. The modified variables are used when a *new* command prompt is opened.

## 1.3 Client scenario

In *Client* scenario, Python calls remote enabled ABAP function module (FM)<sup>1</sup> via SAP RFC protocol, as shown in *Introduction*. To introduce the functionality, we will start with an three *examples*, then show some *details* of the *Connection*, and finally cover some *implementation details*.

### 1.3.1 Examples

To create a connection, construct a *Connection* object and pass the credentials that should be used to open a connection to an SAP backend system.

```
>>> from pyrfc import Connection
>>> conn = Connection(user='me', passwd='secret', ashost='10.0.0.1', sysnr='00', client='100')
```

For the examples we usually store the logon information in a config document (*sapnwrfc.cfg*) that is read with *ConfigParser*. Thus, if the logon information is stored in a dictionary, we may construct a *Connection* instance by *unpacking* the dictionary, e.g.

```
>>> params = {'user': 'me', 'passwd': 'secret', 'ashost': '10.0.0.1', 'sysnr': '00', 'client': '100'}
>>> conn = Connection(**params)
```

#### Example *clientStfcStructure.py*

Lets do a remote function call with a more complex set of parameters.

A function module knows four types of parameters:

1. IMPORT parameters, set by the client.
2. EXPORT parameters, set by the server.
3. CHANGING parameters, set by the client, can be modified by the server.
4. TABLE parameters, set by the client, can be modified by the server.

A simple example of an RFC with different parameter types can be found in the file *clientStfcStructure.py* in the *examples/* directory. The FM *STFC\_STRUCTURE* uses the IMPORT parameter *IMPORTSTRUCT*, copies it to the EXPORT parameter *ECHOSTRUCT*, then modifies it and appends it to the TABLE parameter *RFCTABLE*. Furthermore, it fills the EXPORT parameter *RESPTEXT* with some system/call information.

The parameter *IMPORTSTRUCT* is of type *RFCTEST*, which contains 12 fields of different types. We fill these fields with example values (ll. 7-22). (Note: A comment after each fields tells something about the ABAP datatype.)

```
imp = dict(
    RFCINT1=0x7f, # INT1: Integer value (1 byte)
    RFCINT2=0x7ffe, # INT2: Integer value (2 bytes)
    RFCINT4=0x7fffffff, # INT: integer value (4 bytes)
    RFCFLOAT=1.23456789, # FLOAT

    RFCCHAR1=u'a', # CHAR[1]
    RFCCHAR2=u'ij', # CHAR[2]
    RFCCHAR4=u'bcde', # CHAR[4]
    RFCDATA1=u'k'*50, RFCDATA2=u'l'*50, # CHAR[50] each

    RFCTIME=datetime.time(12,34,56), # TIME
    RFCDATE=datetime.date(2012,10,03), # DATE
```

<sup>1</sup> To be invoked externally, the function module needs to be remote-enabled. For the sake of readability, we will use the shorter term (“FM”) throughout the text.



----- ( Structure of RFCTEST (n/uc_length=144/264) --							
NAME	FIELD_TYPE	NUC_LENGTH	NUC_OFFSET	UC_LENGTH	UC_OFFSET	DECIMALS	TYPE_DESCRIPTION
RFCFLOAT	RFCTYPE_FLOAT	8	0	8	0	16	None
RFCCHAR1	RFCTYPE_CHAR	1	8	2	8	0	None
RFCINT2	RFCTYPE_INT2	2	10	2	10	0	None
RFCINT1	RFCTYPE_INT1	1	12	1	12	0	None
RFCCHAR4	RFCTYPE_CHAR	4	13	8	14	0	None
RFCINT4	RFCTYPE_INT	4	20	4	24	0	None
RFCHEX3	RFCTYPE_BYTE	3	24	3	28	0	None
RFCCHAR2	RFCTYPE_CHAR	2	27	4	32	0	None
RFCTIME	RFCTYPE_TIME	6	29	12	36	0	None
RFCDATE	RFCTYPE_DATE	8	35	16	48	0	None
RFCDATA1	RFCTYPE_CHAR	50	43	100	64	0	None
RFCDATA2	RFCTYPE_CHAR	50	93	100	164	0	None
----- ( Structure of RFCTEST ) -----							
RFCTABLE	RFCTYPE_TABLE	RFC_TABLES	144	264	0		False
----- ( Structure of RFCTEST (n/uc_length=144/264) --							
[...]							
----- ( Structure of RFCTEST ) -----							
ECHOSTRUCT	RFCTYPE_STRUCTURE	RFC_EXPORT	144	264	0		False
----- ( Structure of RFCTEST (n/uc_length=144/264) --							
[...]							
----- ( Structure of RFCTEST ) -----							
RESPTEXT	RFCTYPE_CHAR	RFC_EXPORT	255	510	0		False
-----							

Once again some remarks:

1. The `parameter_type` and `field_type` are not the ABAP types (that were given as a comment in the first example), but the type names given by the C connector. For more details on the type conversion, see the [technical details](#).
2. Most of the information presented here is not relevant for client usage. The important values are:
 

**FunctionDescription.parameters** name, `parameter_type`, `direction`, `nuc_length` (in case of fixed length strings or numeric strings), `decimals` (in case of decimal types – `RFCTYPE_BCD`), and `optional`.

**TypeDescription.fields** name, `field_type`, `nuc_length` (in case of fixed length strings or numeric strings), and `decimals` (in case of decimal types – `RFCTYPE_BCD`).
3. Later, in the [Server scenario](#) usage, the classes `FunctionDescription` and `TypeDescription` will be used again when installing a function on a Python server.

## Errors

If something goes wrong while working with the RFC functionality, e.g. invoking a function module that does not exist in the backend, an error is raised:

```
>>> python clientPrintDescription.py STFC_STRUCTURES
... An error occurred.
... [...]
... pyrfc._exception.ABAPApplicationError: Error 5: [FU_NOT_FOUND] ID:FL Type:E Number:046 STFC_S
```

For further description see [Errors](#).

## Example clientIDocUnit.py

**Warning:** The background protocol (bgRFC) is not working in the current version. Please use only tRFC/qRFC protocols.

Certain operations, e.g. sending IDocs, are not possible with the RFC protocol. Rather, a protocol with transactional guarantees has to be used. The first transactional protocols were tRFC (transactional RFC) and qRFC (queued RFC). Afterwards, bgRFC (background RFC) were introduced. All these protocols have in common that they group one or more FM invocations as one *logical unit of work* (LUW). Consequently, a `Connection` object offers various methods to work with such *units*.

Working with units is as follows:

1. Initialize a unit by using `initialize_unit()`. The method returns a unit descriptor, which is used later on. When initializing the unit, decide whether to use the bgRFC protocol (default) or the tRFC or qRFC protocol by setting `background=False`.
2. The next step is to create the unit in the backend system, prepare the invocation of one or more RFC in it and submit the unit to the backend. All this functionality is provided by `fill_and_submit_unit()`. The method takes two required parameters. The first one is a unit descriptor as returned by `initialize_unit()`. The second one is a list of RFC descriptions that should be executed in the unit. A RFC descriptions consists of a tuple with the name of the FM as the first element and a dictionary describing the function container as the second element.
3. If `fill_and_submit_unit()` ended successfully, i.e. without raising an exception, the unit should be confirmed by `confirm_unit()`. In case there is a problem with the unit, it can be deleted in the backend system by calling `destroy_unit()`.

The current state of a unit can be – in case of units using the bgRFC protocol – retrieved by `get_unit_state()`.

The example script `clientIDocUnit.py` provides examples for sending iDocs. The script was inspired by `iDocClient.c` of *Schmidt and Li (2009c, pp. 2ff)*, but omits the implementation of client side features that assure atomic execution (see also next section).

---

**Note:** Use transaction WE05 to see the IDocs recorded in the SAP backend.

---

---

**Note:** In case you are using queued units (qRFC), use transaction SMQR to register a new queue. In transaction SMQ2 (qRFC monitor) you see the incoming calls. Note that it is possible to send the unit to a non-registered queue name. It will be held with status `ready` in the monitor until it is deleted or the queue registered. For further information, see [qRFC Administration](#).

---

## Assuring atomic execution

In order to assure that the unit is executed exactly once, it is of great importance that the **end system** on the client side initiates the confirmation. Citing Schmidt and Le (sapnwrhc.h, l. 1361ff) with modifications:

After [fill\_and\_submit\_unit()] returned successfully, you should use this function to cleanup the status information for this unit on backend side. However, be careful: if you have a three-tier architecture, don't bundle Submit and Confirm into one single logical step. Otherwise you run the risk, that the middle tier (the NW RFC lib) successfully executes both, the Submit and the Confirm, but on the way back to the first tier an error occurs and the first tier can not be sure that the unit was really executed in the backend and therefore decides to re-execute it. This will now result in a duplicate execution in the backend, because the Confirm step in the first try has already deleted the UID in the backend, and consequently the backend is no longer protected against re-execution of this



UID. In a three-tier architecture, the first tier should trigger both steps separately: first the Submit, and after it knows that the Submit was successful, the Confirm.

Also in case the Confirm runs into an error, [...] try the Confirm again at a later point [.]

Further details to this issue can be found in *Schmidt and Li (2009c, pp. 4-5)*.

## 1.3.2 Configuration of a connection

Upon construction, a `Connection` object may be configured in various ways by passing a `config` parameter.

```
>>> conn = Connection(config = {'keyword': value, ...}, **params)
```

The following keywords for the config dictionary are possible:

### `rstrip`

ABAP allows two different ways to store strings: A fixed length string type `C` and a dynamic length string type `STRING`. Strings of type `C` are padded with blanks, if the content is shorter than the predefined length. In order to unify the connector's behavior regarding strings, the `rstrip` option was introduced. If set to `True`, all strings are right-stripped before being returned by an RFC call.

*Default: True*

### `return_import_params`

Usually, you do not need the `IMPORT` parameters in the result of `Connection.call()`. If `return_import_params` is set to `False`, parameters of type `IMPORT` are filtered out.

*Default: False*

---

**Note:** All the parameters are public object attributes, i.e. they can be modified after the object's construction.

---

## 1.3.3 Selected Connection methods

Besides the mentioned methods in the examples, the `Connection` offers some basic methods for working with a connection:

---

<code>Connection.ping</code>
<code>Connection.reset_server_context</code>
<code>Connection.get_connection_attributes</code>
<code>Connection.close</code>

---

## 1.3.4 Technical details

This section describes the *Data types* and the *Data transmission*.

### Data types

A remote function call executes ABAP code, which works with parameters that have an ABAP data type. Hence, when you look at the metadata description you will find ABAP data types for the parameters.

The Python connector does not provide ABAP data types to be instantiated and used within Python code, but rather converts between ABAP data types and Python built-in types.

Type Category	ABAP	Meaning	RFC	Python	Remark
numeric	I	Integer (whole number)	INT	int	Internal 1 and 2 byte integers (INT1, INT2) are also mapped to int
numeric	F	Floating point number	FLOAT	float	
numeric	P	Packed number / BCD number	BCD	Decimal	
character	C	Text field (alphanumeric characters)	CHAR	unicode	
character	D	Date field (Format: YYYYMMDD)	DATE	date-time.date	
character	T	Time field (Format: HHMMSS)	TIME	date-time.time	
character	N	Numeric text field (numeric characters)	NUM	unicode	
hexadecimal	X	Hexadecimal field	BYTE	str [bytes]	
variable length	STRING	Dynamic length string	STRING	Unicode	
variable length	XSTRING	Dynamic length hexadecimal string	BYTE	str [bytes]	

Further [details on predefined ABAP types](#) are available online.

The Python representation of a parameter is a simple key-value pair, where the key is the name of the parameter and the value is the value of the parameter in the corresponding Python type. Beside the mentioned types, there are tables and structures:

- A structure is represented in Python by a dictionary, with the structure fields' names as dictionary keys.
- A table is represented in Python by a list of dictionaries.

For an example see [Example clientSfcsStructure.py](#).

## Data transmission

The data transmission in the C connector takes place as follows: If you want to invoke an RFC, a function container is constructed from the metadata description of the RFC. The function container is a memory structure to which the parameters are written. Then the RFC is invoked and the function container is passed to the backend system. The backend system now executes the RFC on the given function container, i.e. it reads some values from the function container and writes other values to it. Finally, the function container is passed back to the C connector.

This has some important consequences:

- There is no technical distinction between input and output values.
- In the metadata description, each parameter is classified as IMPORT, EXPORT, CHANGING, and TABLES. Hence, there is a convention regarding which parameters are set when the RFC is invoked and which parameters are filled or changed after the RFC's execution.
- It is possible, though not good practice, to set the output values (i.e. parameter of type EXPORT) when invoking an RFC. Similarly, it is possible that an RFC will modify the input values (parameters of type IMPORT). However, a well written RFC will not manipulate the input values and initialize the output values to a default value.

## 1.4 Server scenario

Server usage refers to a situation, in which a Python script serves RFC requests that are sent by an SAP backend system.

To illustrate the usage, we will first show an *Example serverStfcConnection.py*, then describe some basic aspects of the *Server* and finally cover some *Advanced topics*.

**Note:** For this section, we assume previous knowledge from the *Client scenario* section. Furthermore, the server related article by *Schmidt and Li (2009b)* is highly recommended.

**Warning:** Server functionality is currently not working with Python 32bit under Windows.

### 1.4.1 Example serverStfcConnection.py

Creating a Python RFC server consists basically of two parts:

1. A `Server` object that stores metadata about the functions that it will serve, and that later registers at a gateway.
2. For each served function, a callback function (or: implementing function) has to be provided.

A simple example for a server can be found in the file `serverStfcConnection.py` in the `examples/` directory. This server offers a function with the same metadata as `STFC_CONNECTION`, but will return a slightly modified version as the usual ABAP implementation.

The metadata for `STFC_CONNECTION` consists of

- one `IMPORT` parameter – `REQTEXT` – and
- two `EXPORT` parameters – `ECHOTEXT` (usually a copy of `REQTEXT`) and `RESPTEXT` (usually some connection/system details).

#### STFC\_CONNECTION callback function

Lets look at our callback function (ll. 9-14), that implements the server logic:

```
config = ConfigParser()
config.read('sapnwrfc.cfg')

# Callback function
def my_stfc_connection(request_context, REQTEXT=""):
    return {
```

The callback function takes two parameters. The first one, `request_context` contains call-specific information and is obligatory for any callback function. Afterwards, the parameters depend on the function's metadata description. In our case, there is one `IMPORT` parameter that is expected by the callback function.

The callback function fills the value for the two `EXPORT` parameters and returns them in a dictionary.

#### STFC\_CONNECTION Server

The `Server` class offers server related functionality. In the example, its usage is found in lines 23-27.

```
params_connection = config._sections['connection']
conn = Connection(**params_connection)
func_desc_stfc_connection = conn.get_function_description("STFC_CONNECTION")

# Instantiate server with gateway information for registering, and
```

- First, a `Server` object is created and gateway parameters are passed.
- Second, a function is installed via `install_function()`. The parameters are a `FunctionDescription` object and a callback function.

- Finally, the server serves requests by invoking the `serve()` method.

A remark regarding the second point: The function description for our callback function is retrieved from an SAP system in lines 17-20.

```
'RESPTEXT': u"Python server here. Connection attributes are: "  
            u"User '{user}' from system '{sysId}', client '{client}', "  
            u"host '{partnerHost}'".format(**request_context['connection_attributes'])  
}
```

Retrieving functions descriptions in such a way is convenient for various reasons (cf. *Schmidt and Li (2009b, p. 2)*). However, hard coding of function descriptions is possible (cf. *Example serverFunctionDescription.py*).

## Registering the server

Registering the server needs some preparation in the SAP backend. Configuration of RFC connections is handled in transaction SM59. Create a new RFC destination (e.g. `PYTHON_SVR_DEST`) of type T and choose under technical settings:

- Activation type: Registered Server Program
- Registered Server Program: Program ID <your program ID>

The program ID is used when instantiating a Server object.

## Invoking the server

A simple approach to invoke our server is to

1. log on to an SAP backend system,
2. use transaction SE37,
3. test/execute the function module `STFC_CONNECTION`, and
4. set RFC target `sys.` to the RFC destination of the server (e.g. `PYTHON_SVR_DEST`).

### 1.4.2 Server

For server usage, the Python connector offers the class `Server`. An object is instantiated with gateway parameters. The server will register at this gateway before serving requests.

Gateway parameters are:

**GWHOST** The name of the gateway host

**GWSERV** The name of the gateway server

**PROGRAM\_ID** The name under which the Python connector will register at the gateway. This corresponds to an RFC destination in transaction SM59 of type “T” in registration mode.

**TRACE** Sets the trace level, cf. the documentation of `RfcSetTraceLevel` in `sapnwrfc.h` of the C connector.

**SAPROUTER** An SAP router string

Furthermore, the server accepts a `config` parameter.

## Config

Upon construction, a `Server` object may be configured in various ways by passing a `config` parameter.

## `rstrip`

ABAP allows two different ways to store strings: A fixed length string type `C` and a dynamic length string type `STRING`. Strings of type `C` are padded with blanks, if the content is shorter than the predefined length. In order to unify the connectors behavior regarding strings, the `rstrip` option was introduced. If set to `True`, all strings are right stripped before being passed to the callback function.

*Default: True*

## Server functions

The `Server.install_function()` installs a function in the server. It expects two parameters: a `FunctionDescription` object for the metadata description and a callback function that implements the server logic.

The callback function will be called if the gateway receives an RFC call for the given `FunctionDescription` and if the server object is serving requests (`Server.serve()`). In this case, the callback function is called with the following parameters:

**request\_context** A dictionary with the following key:

**connection\_attributes** A dictionary with connection attributes of the client. The keys are the same as returned by `Connection.get_connection_attributes()`, excluding `alive` and `active_unit`. As done by `Connection.get_connection_attributes()`, the values are right stripped strings. These connection attributes may be used for authorization checks.

For a future release, information about active parameters will be given here.

**<PARAMETERS>** All `IMPORT`, `CHANGING`, and `TABLE` parameters of the `FunctionDescription`.

## 1.4.3 Advanced topics

### Raising exceptions

An external server program is allowed to throw errors as a usual ABAP function module. To do so, a certain type of exception is raised.

Error type	Corresponds to ABAP statement	In Python triggered via	Arguments to pass	Effect on connection	Effect in the back end (ABAP)
ABAP exception	RAISE <exception key>	raise ABA-PApplication-Error(...)	key	Remains open	SY-SUBRC is set corresponding to the exception key in the EXCEPTIONS clause.
ABAP exception with details	MESSAGE ... RAISING <exception key>	raise ABA-PApplication-Error(...)	key, msg_type, msg_class, msg_number, msg_v1-v4	Remains open	As above. The following fields are filled: SY-MSGTY, SY-MSGID, SY-MSGNO, and SY-MSGV1-V4.
ABAP message	MESSAGE ...	raise ABAPRun-timeError(...)	msg_type, msg_class, msg_number, msg_v1-v4	Is closed	SY-SUBRC is set corresponding to the SYSTEM_FAILURE key in the EXCEPTIONS clause and the SY-MSG fields are filled as above.
System failure		raise ExternalRun-timeError(...)	message	Is closed	SY-SUBRC is set corresponding to the SYSTEM_FAILURE key in the EXCEPTIONS clause and the parameter specified in the MESSAGE addition is filled.

Other exceptions are not permitted. For further details on error raising cf. *Schmidt and Li (2009b, pp. 6ff)*.

Note that the arguments have a maximum length. If a longer string is passed, only the first valid number of characters will be used.

- key: 128 chars
- message: 512 chars
- msg\_type: 1 char
- msg\_class: 20 chars
- msg\_number: 3 chars
- msg\_v1-v4: 50 chars each

## Authorization

At the current state of the Python connector, an authorization check has to be implemented in the callback function by evaluating the connection attributes found in `request_context['connection_attributes']` (cf. *Server functions*).

## Hard-coded function descriptions

In some cases, it is not possible to retrieve the metadata from an SAP backend system (cf. *Schmidt and Li (2009c, pp. 9ff)*). For these situations, objects of `FunctionDescription` and `TypeDescription` can be hard-coded.

The required methods are

---

```
FunctionDescription.add_parameter
TypeDescription.add_field
```

---

**Example serverFunctionDescription.py**

Similar to the example `hardCodedServer.c` of the C connector, in the file `serverFunctionDescription.py` in the `examples/` directory, a function description – and included type descriptions – are constructed manually. In line 3-15 we define a `TypeDescription` object consisting of three fields:

```
animals = TypeDescription("ANIMALS", nuc_length=20, uc_length=28)
animals.add_field(name=u'LION', field_type='RFCTYPE_CHAR',
                  nuc_length=5, uc_length=10,
                  nuc_offset=0, uc_offset=0
                )
animals.add_field(name=u'ELEPHANT', field_type='RFCTYPE_FLOAT', decimals=16,
                  nuc_length=8, uc_length=8,
                  nuc_offset=8, uc_offset=16,
                )
animals.add_field(name=u'ZEBRA', field_type='RFCTYPE_INT',
                  nuc_length=4, uc_length=4,
                  nuc_offset=16, uc_offset=24
                )
```

Afterwards, a `FunctionDescription` object is created and several fields are added.

```
func_desc = FunctionDescription("I_DONT_EXIST")
func_desc.add_parameter(name=u'DOC', field_type='RFCTYPE_INT',
                        direction='RFC_IMPORT',
                        nuc_length=4,
                        uc_length=4
                      )
func_desc.add_parameter(name=u'CAT', field_type='RFCTYPE_CHAR',
                        direction='RFC_IMPORT',
                        nuc_length=5,
                        uc_length=10
                      )
func_desc.add_parameter(name=u'ZOO', field_type='RFCTYPE_STRUCTURE',
                        direction='RFC_IMPORT',
                        nuc_length=20,
                        uc_length=28,
                        type_description=animals
                      )
func_desc.add_parameter(name=u'BIRD', field_type='RFCTYPE_FLOAT',
                        direction='RFC_IMPORT',
                        nuc_length=8,
                        uc_length=8,
                        decimals=16
                      )
func_desc.add_parameter(name=u'COW', field_type='RFCTYPE_CHAR',
                        direction='RFC_EXPORT',
                        nuc_length=3,
                        uc_length=6
                      )
func_desc.add_parameter(name=u'STABLE', field_type='RFCTYPE_STRUCTURE',
                        direction='RFC_EXPORT',
                        nuc_length=20,
                        uc_length=28,
                        type_description=animals
                      )
func_desc.add_parameter(name=u'HORSE', field_type='RFCTYPE_INT',
                        direction='RFC_EXPORT',
                        nuc_length=4,
                        uc_length=4
                      )
```

## 1.5 Security

Plain RFC connections are mainly used for prototyping, while in production secure connections are required. For more information on RFC security see:

- [Security on SAP Service Marketplace](#)
- [RFC Security Best Practices on SAP SCN](#)
- [Secure Network Communication \(SNC\) - SAP Help](#)
- [SNC User Guide](#)

SAP NW RFC Library supports plain and secure connection with following authentication methods:

- Plain with user / password
- SNC
  - with User PSE
  - with X509

NW ABAP servers support in addition:

- SAP logon tickets
- Security Assertion Markup Language (SAML)

Assuming you are familiar with abovementioned concepts and have ABAP backend system configured for SNC communication, here you may find connection strings examples, for testing plain and secure RFC connections, with various authentication methods.

### 1.5.1 Authentication

#### Plain with user / password

The simplest and least secure form of the user authentication.

```
ABAP_SYSTEM = {
    'user': 'demo',
    'passwd': 'welcome',

    'name': 'I64',
    'client': '800',
    'ashost': '10.0.0.1',
    'sysnr': '00',
    'saprouter': SAPROUTER,
    'trace': '3'
}

c = get_connection(ABAP_SYSTEM) # plain
```

#### SNC with User PSE

User PSE is used for opening the SNC connection and the same user is used for the authentication (logon) in NW ABAP backend. Generally not recommended, see [SAP Note 1028503 - SNC-secured RFC connection: Logon ticket is ignored](#)

#### Prerequisites

- SNC name must be configured for the ABAP user in NW ABAP system, using transaction SU01
- SAP Single Sign On must be configured on a client and the user must be logged in on a client.



The screenshot shows the 'Display User' SAP GUI. The 'User' field is 'USTEST'. The 'Last Changed On' field shows 'D037732' and '28.01.2014 14:29:21'. The 'Status' is 'Saved'. The 'SNC' tab is selected, showing 'SNC Status' and 'SNC data'. The 'SNC data' section shows 'SNC name' as 'p:CN=USTEST, O=SAP-AG, C=DE' and a checked box for 'Canonical name determined'. There is also an unchecked box for 'Unsecure communication permitted (user-specific)'.

```

ABAP_SYSTEM = {
  'snc_partnername': 'p:CN=I64, O=SAP-AG, C=DE',
  'snc_lib': 'C:\\Program Files (x86)\\SECUDE\\OfficeSecurity\\secude.dll',

  'name': 'I64',
  'client': '800',
  'ashost': '10.0.0.1',
  'sysnr': '00',
  'saprouter': SAPROUTER,
  'trace': '3'
}

c = get_connection(ABAP_SYSTEM)

```

In this example the SNC\_LIB key contains the path to security library (SAP cryptographic library or 3rd party product). Alternatively, the SNC\_LIB can be set as the environment variable, in which case it does not have to be provided as a parameter for opening SNC connection.

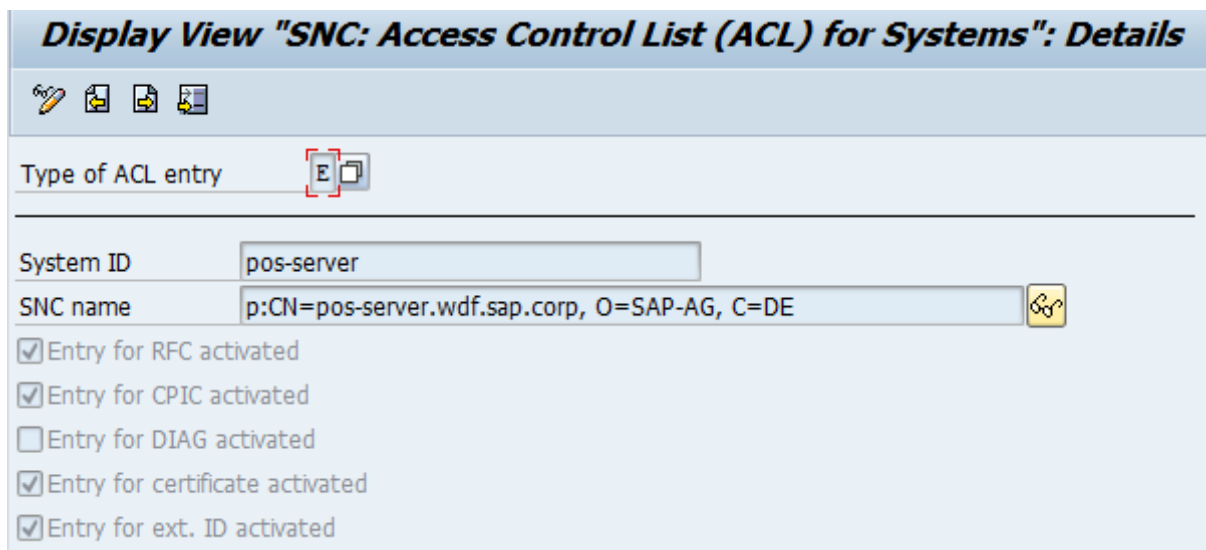
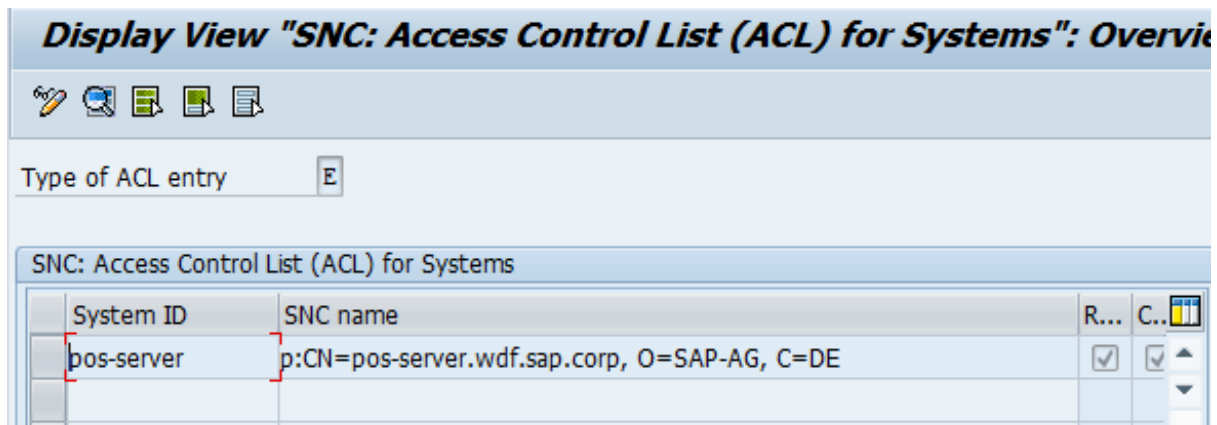
## SNC with X509

The client system PSE is used for opening SNC connection and forwarding user X509 certificate to NW ABAP backend system, for authentication and login.

### Prerequisites

- The user does not have to be logged into the client system, neither the Single Sign On must be configured on a client
- The trusted relationship must be established between the NW ABAP backend and the client system.
- The client system must be registered in the NW ABAP backend Access Control List (ACL), using transaction SNC0

- Keystores are generated on a client system, using SAP cryptography tool *SAPGENPSE* and the environment variable *SECUDIR* points to the folder with generated keystores



- User X509 certificate must be mapped to ABAP NW backend user, using transaction *EXTID\_DN*

The same connection parameters as in a previous example, with X509 certificate added.

```
ABAP_SYSTEM = {
    'snc_partnername': 'p:CN=I64, O=SAP-AG, C=DE',
    'snc_lib': 'C:\\Program Files (x86)\\SECUDE\\OfficeSecurity\\secude.dll',






    'x509cert': 'MIIDJjCCAtCgAwIBAgIBNzA ... NgalgcTJf3iUjZ1e5Iv5PLKO',

    'name': 'I64',
    'client': '800',
    'ashost': '10.0.0.1',
    'sysnr': '00',
    'saprouter': SAPROUTER,
    'trace': '3'
}

c = get_connection(ABAP_SYSTEM)
```

See [SAP Help](#) for more information.

**Display View "Assignment of External ID to Users": Overview**











External ID type  DN of Certificate (X.500)


Assignment of External ID to Users

	H.. External ID	User	Act.
<input type="checkbox"/>	CN=Hugo, OU=CSI, O=SAP Trust Community, C=DE	USTEST	<input checked="" type="checkbox"/>


**Display View "Assignment of External ID to Users": Details**

External ID type  DN of Certificate (X.500)


External ID  

Seq. No.

User  

Min. date

☒ Activated

Issuer  

## 1.6 Building from source

SAP NW RFC Library shall be installed as described in the *Installation*. Instead of downloading and installing precompiled egg, you need to prepare the toolchain and clone the `pyrfc` repository, so that you can build the distribution release code and documentation.

### 1.6.1 Toolchain preparation

#### Linux platform

- *Install Python*
- Install `easy_install` as described in *pyrfc installation*
- *Install SAP NW RFC Library*
- To get any software from the Git source control system the Git client is required as well, use whatever your distribution has
- Install Cython. Versions tested so far are 0.17.2, 0.19.0 and 0.20.0, other are expected to work as well.
- The system must contain a `gcc` compiler as well as development header and library files as provided by your distribution.

#### Windows platform

- *Install Python*
- Install `easy_install` as described in *pyrfc installation*
- *Install SAP NW RFC Library*
- To get any software from the Git source control system the Git client is required as well. Download and install from <http://code.google.com/p/msysgit/downloads/list?can=3>. During installation specify that Git runs out of the Bash shell as you may need that shell later on.
- Install Cython, using *Windows installer*, see also <http://wiki.cython.org/64BitCythonExtensionsOnWindows>
- Download and install the compiler toolchain, tested on Windows 7 32 and 64 bit platforms
  - MS VisualStudio2008 Express Edition
  - Microsoft Windows SDK for Windows 7 and .NET Framework 3.5 SP1

### 1.6.2 Building the code

To build eggs for different Python versions, install these versions on your system and create a virtual environment for each of these versions, for example:

```
virtualenv --python=<PATH e.g. c:\Python27\python.exe OR  
/usr/bin/python2.7> ...
```

Otherwise, follow the example below.

#### Linux platform

Clone the repository:

```
git clone https://github.com/SAP/PyRFC
```

Edit `setup.py` and set the `CYTHON_VERSION`

Build the distribution

```
python setup.py clean --all
python setup.py bdist_egg
```

The result is found in the `dist/` directory. The process has to be done on all platforms for which we provide eggs.

## Windows platform

Open the GIT Bash shell and clone the repository.

```
git clone https://github.com/SAP/PyRFC
```

Open the CMD Shell from Microsoft Windows SDK 7.0 and change to cloned `pyrfc` folder.

Edit `setup.py` and set the `CYTHON_VERSION`

Set env variables for the release, use `/x64` for 64 bit and `/x86` for 32 bit:

```
set DISTUTILS_USE_SDK=1
setenv /x64 /release
```

Build the distribution:

```
python setup.py clean --all
python setup.py bdist_egg
```

Check the `pyrfc\dist` folder for a new created egg.

## Virtual Environments

You may have both 32bit and 64bit versions of Python installed on your system and use virtual environments. This is basically possible (e.g. installing the 32bit version on 64 bit system in `C:\Python27_32\`), but beware of modifying the `PATH` variable.

However, the `PATH` variable is modified when using a virtual environment, therefore modify the `Scripts/activate.bat` file with:

```
set SAPNWRFC_HOME=C:\nwrfdc_sdk_x86
set PATH=C:\nwrfdc_sdk_x86\lib\;%PATH%
set PATH=%VIRTUAL_ENV%\Scripts;%PATH%
```

This assures that specific SAP NW RFC Library is used (e.g. 32bit in this example). This is not required for building the distribution, but rather for importing the Python connector.

The build process remains the same, only before building the distribution, you need to activate the virtual environment and assure that library paths are correct in `setup.py`.

## Python 3

Prerequisites for building on Python 3, tested on Linux Mint and Ubuntu

```
sudo apt-get install python3-setuptools python3-dev python-configparser
sudo easy_install3 pip
sudo pip3 install cython sphinx ipython
```

### 1.6.3 Building the documentation

Ensure that the lib directory of the SAP NW RFC library is in your PATH environment.

Change into the doc directory and type:

```
make clean
make html
```

The result is found in `_build/html` and for other options call `make`.

- If you get an error *'sphinx-build' is not recognized as an internal or external command, operable program or batch file* on calling `make html`, install `sphinx`
- If you have DLL import errors (Windows), check the lib directory of the SAP NW RFC Library PATH env variable.

The docu is hosted on GitHub Pages, a proprietary solution where a git branch `gh-pages` is created as an orphan and the output of the documentation build process (`_build/html`) is stored in that branch. GitHub then serves these files under a special `/pages/` url.

To update GitHub Pages, copy everything under `_build/html` and overwrite the existing files in the `gh-pages` branch root.

```
cp _build/html ~/tmp
git checkout gh-pages
rm -Rf *.html *.js *.egg build doc _* pyrfc* *.inv .buildinfo
cp -R ~/tmp/_build/html/. .
```

---

**Note:** An additional file `.nojekyll` is placed in `gh-pages` to disable the default GitHub processing which breaks sphinx style folders with leading underscores.

`gh-pages` updates are a bit inconvenient, check if this answer helps <http://stackoverflow.com/questions/4750520/git-branch-gh-pages>

---

## 1.7 Remarks

### 1.7.1 RFC call results differ from expected

The response when invoking ABAP function module directly via transaction SE37 may be different from the response when called via Python connector. Typically leading zeros are not shown in SE37 but sent to Python when remote FM is called from Python.

This is surely a bug and in general there are few possible causes: the `pyrfc` package, the *SAP NW RFC Library*, or the function module itself. Bugs in *SAP NW RFC Library* are unlikely, in `pyrfc` less likely and the cause is usually the implementation of the ABAP FM, not respecting technical restriction described [here](#).

Conversion (“ALPHA”) exists are triggered in SAP GUI and in SE37 but not when the ABAP FM is called via SAP RFC protocol, therefore the rectifying logic shall be implemented in the FM. The behaviour can be also caused by different authorization levels, when ABAP FM is invoked locally or remotely, via SAP RFC protocol.

Please try to assure that RFC is working (e.g. testing with the C connector), before reporting the problem.

### 1.7.2 Multiple threads

If you work with a multiple thread environment, assure that each thread has its own `Connection` object. The reason is that during the RFC calls, the Python global interpreter lock (GIL) is released.

### 1.7.3 Open, valid, and alive connections

A connection may be closed by the client (e.g. via `Connection.close()`) or by the server (e.g. if an ABAP message is raised). The connection object maintains a object variable `alive` to record the actual state. However, the variable is of internal use only, as the connection object *will try to reopen the connection, if needed*. This leads to the – seemingly strange – situation that an explicitly closed connection will allow you to call, e.g. `Connection.ping()`.

Implementation remark: It is not possible to query the actual state of the connection in a reliable manner. Although *SAP NW RFC Library* offers the function `RfcIsConnectionHandleValid()`, it will only return *False* if the connection was closed by the client, not if it was closed from the server side. Therefore, an explicit object variable is kept.

## 1.8 Bibliography

### 1.8.1 Schmidt and Li (2009a)

Improve communication between your C/C++ applications and SAP systems with SAP NetWeaver RFC SDK - [Part 1: RFC Client Programming](#), *SAP Professional Journal*, pp 1-16 (originally published in November 2007)

### 1.8.2 Schmidt and Li (2009b)

Improve communication between your C/C++ applications and SAP systems with SAP NetWeaver RFC SDK - - [Part 2: RFC Server Programming](#), *SAP Professional Journal*, pp 1-13 (originally published in January/February 2008)

### 1.8.3 Schmidt and Li (2009c)

Improve communication between your C/C++ applications and SAP systems with SAP NetWeaver RFC SDK - [Part 3: Advanced Topics](#), *SAP Professional Journal*, pp 1-18 (originally published in March 2008)





---

## API documentation

---

### 2.1 `pyrfc`

The `pyrfc` package.

#### 2.1.1 Connection

#### 2.1.2 Server

#### 2.1.3 `FunctionDescription`

---

**Note:** Actually, the `FunctionDescription` class does not support exceptions.

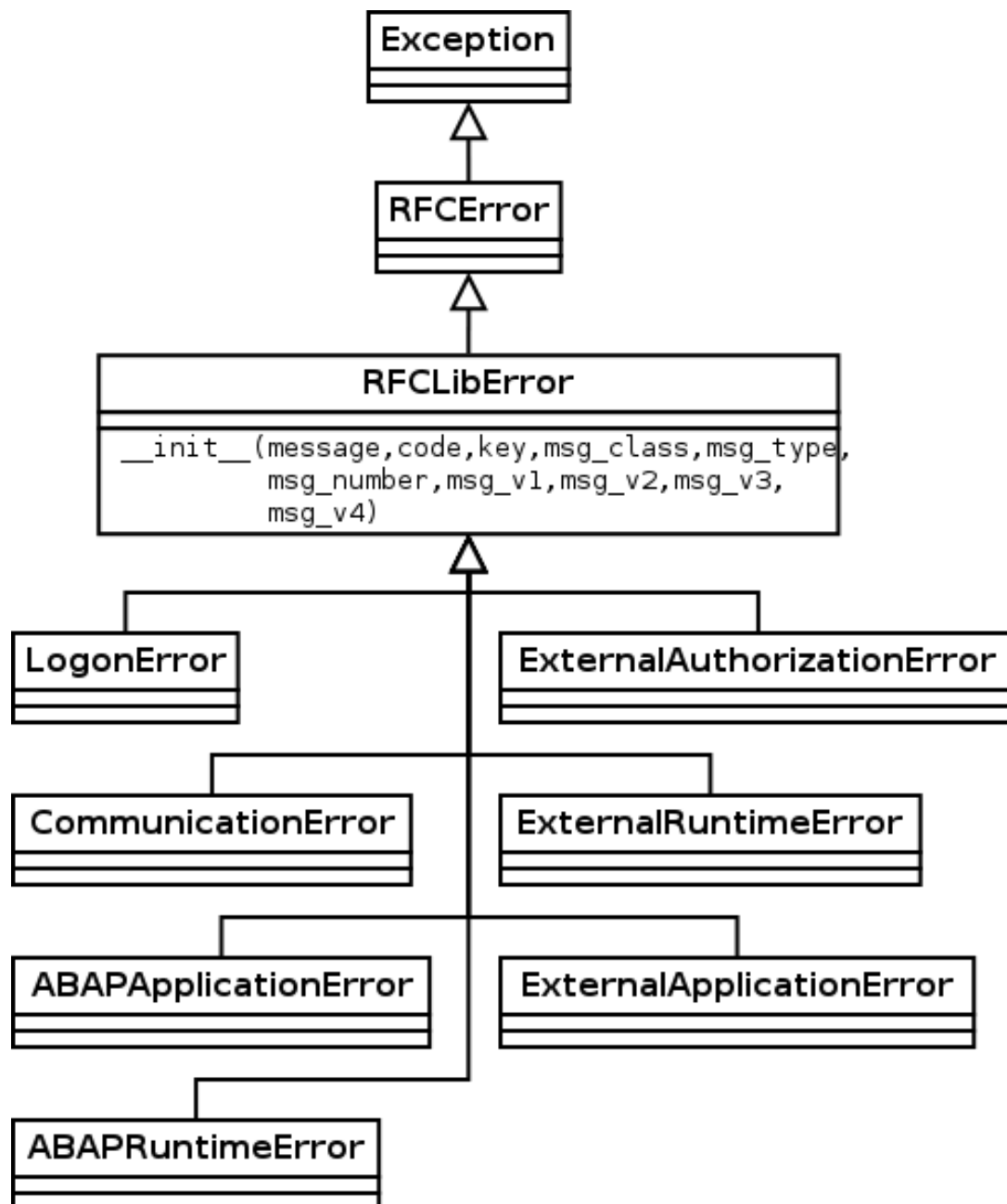
---

#### 2.1.4 `TypeDescription`

#### 2.1.5 Errors

If a problem occurs in the Python connector or in an underlying component (e.g. C connector, SAP system, ABAP code, ...), an exception is raised. The class of the exception indicates where the problem occurred.

1. `RFCErrors`: This error is raised, if a problem occurred in the Python connector.
2. `RFCLibError`: This error is raised, if a problem occurred in the C connector.
3. All other errors represent errors with the RFC call to the SAP backend system. For these errors, the `errorInfo` struct of the C connector is wrapped, e.g. for a given exception `e`, the error code is available in `e.code`. The class of the error depends on the group of the error.



### Error types, codes, groups, and classes

*Schmidt and Li (2009a)* describe four possible *error types* on the basis of the return code (i.e. *error code*) of a RFM invocation:

- ABAP exception,
- system failure,
- ABAP messages, and
- communication failure.

However, there are in total roughly 30 possible return codes that indicate some kind of error. As each error information struct provides an *error group* information with seven possible groups, which was taken as the basis for the exception *classes*.

The following table should facilitate the matching between the different error representations.

type (SPJ)	code [numeric] (C)	group (C)	class (Python)
ABAP exception	RFC_ABAP_EXCEPTION [5]	ABAP_APPLICATION_FAILURE	ABAPApplicationError
system failure	RFC_ABAP_RUNTIME_FAILURE [3]	ABAP_RUNTIME_FAILURE	ABAPRuntimeError
ABAP message	RFC_ABAP_MESSAGE [4]	ABAP_RUNTIME_FAILURE	ABAPRuntimeError
communication failure	RFC_COMMUNICATION_FAILURE [1]	RECOMMUNICATION_FAILURE	CommunicationError
	RFC_LOGON_FAILURE [2]	LOGON_FAILURE	LogonError



---

## Change log

---

### 3.1 Change log

1.9.4 (Development) - RfcGetVersion API added - More docu - Unit tests

#### 3.1.1 1.9.3 (2014-01-26)

- Fixed Date wrap bug (reported by Flavio Marinone)
- Documentation rework

#### 3.1.2 1.9.2 (2012-11-26)

- Fixed BCD wrap bug (reported by Thomas Marcon)

#### 3.1.3 1.9.1 (2012-10-02)

- Added server usage
  - Server class (wraps server related activities)
  - FunctionDescription and TypeDescription as classes for improved metadata handling.
- Added support for transactional, queued, and background RFC (tRFC, qRFC, bgRFC) for client usage.
- Renamed exceptions in accordance with PEP08:
  - Exception names changed to Error (e.g. ExternalRuntimeError renamed to ExternalRuntimeError).
  - Rfc now uppercase, e.g. RFCLibError.
  - Abap now uppercase, e.g. ABAPRuntimeError.
- Connection class
  - removed `__init__` optional parameter *strip*, added *config* dictionary as optional parameter.
  - renamed `connectionInfo` to `get_connection_attributes()`
  - renamed `resetServerContext` to `reset_server_context()`
- RFCTYPE\_BCD fields are now converted to Decimal object (wrap). For filling, you may pass either Decimal objects or floats.

### 3.1.4 1.9 (2012-09-14)

- Restructured exceptions:
  - All exceptions are now directly available in the pyrfc package (e.g. from pyrfc import LogonException)
  - RfcLibException renamed to *RfcLibException*
  - All used exceptions (corresponding to RFC error group classes) inherit now from *RfcLibException* (implication for *LogonException*, *AbapRuntimeException*, *AbapApplicationException*)
  - Constructor for *RfcLibException* changed (first argument is now message), allowing to raise exceptions manually with only one string value
  - *AbapException* deleted.
  - Added external exceptions (*ExternalRuntimeException*, *ExternalApplicationException*, *ExternalAuthorizationException*)
- restructured and added tests (MRFC and STFC function groups).
  - fixed bugfix implications from rev. 1.8.2. (*LogonException* does not have code fields now)
- Added examples (sample/): *printDescription.py*

### 3.1.5 1.8.2 (2012-04-05)

- bugfix: *LogonException* now inherits from *RfcException* instead of *RfcLibException* so it only needs one message argument.

### 3.1.6 1.8.1 (2012-02-27)

- (bugfix in removed module)

### 3.1.7 1.8 (2012-02-07)

- added strip parameter in *Connection* to rstrip whitespaces in rfc results
- added *resetServerContext*
- fixed bug when dealing with xstring results
- added `__bool__` for easy check if connections is alive
- release GIL for *RfcInvoke* and *RfcOpenConnection*

### 3.1.8 1.7 (2011-12-23)

- wrapped *fillVariable* into try..except block to enrich exception information
- Added `_pyrfc.connectionInfo` (*sapnwrfc.RfcGetConnectionAttributes*)

### 3.1.9 1.6 (2011-12-12)

- added bcd conversion
- added performance test
- added *setup.py* test suite

### 3.1.10 1.5 (2011-12-05)

- first buildout configuration that works on Linux and Windows
- rename the C extension to `_pyrfc` and importing it at the root of a new “real” Python package `pyrfc`. This is fully backwards compatible and allows splitting the single module into a C-only part and separate Python modules for better maintainability.
- move pure Python code from the Cython extension module into separate Python modules for maintainability

### 3.1.11 1.4 (2011-11-25)

- start on package documentation
- re-added support for zero dates

### 3.1.12 1.3 (2011-11-22)

- fixed various buffer length issues

### 3.1.13 1.2 (2011-11-18)

- malloc’ed buffer, never too small
- type-specific RFCGetters

### 3.1.14 1.1 (2011-11-16)

- your buffer was too small

### 3.1.15 1.0 (2011-11-11)

- first egg release





---

## Indices and tables

---

- `genindex`
- `search`