example_of_ssh_client_communication

July 20, 2022

1 Communication with a remote Linux machine through SSH

In this notebook, we show how to communicate with a remote machine. This machine was set up using Ubuntu 12 (and with Azure : Create a Virtual Machine Running Linux). The following code is used to get the credentials without printing them on the notebook.

[1]:
```
%load_ext pyensae
%load_ext pyenbc
```

[2]:
```
import pyquickhelper.ipythonhelper as ipy
params={"server":"", "username":"", "password":""}
ipy.open_html_form(params=params,title="credentials",key_save="ssh_remote")
```

[2]:
```
<IPython.core.display.HTML at 0x746b9d0>
```

[3]:
```
password = ssh_remote["password"]
server = ssh_remote["server"]
username = ssh_remote["username"]
```

We are going to use the following magic commands MagicRemote:

[4]:
```
from pyense.remote import ASSHClient
help(ASSHClient)
```

```
Help on class ASSHClient in module pyenbc.remote.ssh_remote_connection:

class ASSHClient(builtins.object)
 |  A simple class to access to remote machine through SSH.
 |  It requires modules
 |  `paramiko <http://www.paramiko.org/>`_,
 |  `pycrypto <https://pypi.python.org/pypi/pycrypto/>`_,
 |  `ecdsa <https://pypi.python.org/pypi/ecdsa>`_.
 |
 |  This class is used in magic command @see me remote_open.
 |  On Windows, the installation of pycrypto can be tricky.
 |  See `Pycrypto on Windows
<http://www.xavierdupre.fr/blog/2014-10-21_nojs.html>`_.
 |  Those modules are part of the `Anaconda
<http://docs.continuum.io/anaconda/pkg-docs.html>`_ distribution.
 |
 |  Methods defined here:
 |
```

```
|  __init__(self, server, username, password)
|      constructor
|
|      @param      server      server
|      @param      username    username
|      @param      password    password
|
|  __str__(self)
|      usual
|
|  close(self)
|      close the connection
|
|  close_session(self)
|      close a session
|
|  connect(self)
|      connect
|
|  dfs_exists(self, path)
|      tells if a file exists on the cluster
|
|      @param      path        path
|      @return                 boolean
|
|      .. versionadded:: 1.1
|
|  dfs_ls(self, path)
|      return the content of a folder on the cluster as a DataFrame
|
|      @param  path        path on the cluster
|      @return             DataFrame
|
|      .. versionadded:: 1.1
|
|  dfs_mkdir(self, path)
|      creates a directory on the cluster
|
|      @param      path        path
|
|      .. versionadded:: 1.1
|
|  dfs_rm(self, path, recursive=False)
|      removes a file on the cluster
|
|      @param      path        path
|      @param      recursive   boolean
|
|      .. versionadded:: 1.1
|
|  download(self, remotepath, localpath)
|      download a file from the remote machine (not on the cluster)
|      @param      localpath     local file
|      @param      remotepath    remote file (it can be a list, localpath is a
```

```
folder in that case)
 |
 |      .. versionchanged:: 1.1
 |          remotepath can be a list of paths
 |
 |  download_cluster(self, remotepath, localpath, merge=False)
 |      download a file directly from the cluster to the local machine
 |      @param      localpath       local file
 |      @param      remotepath      remote file (it can be a list, localpath is
a folder in that case)
 |      @param      merge           True to use getmerge instead of get
 |
 |      .. versionadded:: 1.1
 |
 |  execute_command(self, command, no_exception=False, fill_stdin=None)
 |      execute a command line, it raises an error
 |      if there is an error
 |
 |      @param      command         command
 |      @param      no_exception    if True, do not raise any exception
 |      @param      fill_stdin      data to send on the stdin input
 |      @return                     stdout, stderr
 |
 |      Example of commands::
 |
 |          ssh.execute_command("ls")
 |          ssh.execute_command("hdfs dfs -ls")
 |
 |  exists(self, path)
 |      tells if a file exists on the bridge
 |
 |      @param      path        path
 |      @return                 boolean
 |
 |      .. versionadded:: 1.1
 |
 |  ls(self, path)
 |      return the content of a folder on the bridge as a DataFrame
 |
 |      @param  path        path on the bridge
 |      @return             DataFrame
 |
 |      .. versionadded:: 1.1
 |
 |  open_session(self, no_exception=False, timeout=1.0, add_eol=True,
prompts=('~$', '>>>'), out_format=None)
 |      open a session with method `invoke_shell <http://docs.paramiko.org/en/la
test/api/client.html?highlight=invoke_shell#paramiko.client.SSHClient.invoke_she
ll>`_
 |
 |      @param      no_exception    if True, do not raise any exception in case
of error
 |      @param      timeout         timeout in s
 |      @param      add_eol         if True, the function will add a EOL to the
```

```
sent command if it does not have one
 |      @param      prompts         if function terminates if the output ends by
one of those strings.
 |      @param      out_format      None, plain, html
 |
 |      @example(How to open a remote shell?)
 |      @code
 |      ssh = ASSHClient(   "<server>",
 |                          "<login>",
 |                          "<password>")
 |      ssh.connect()
 |      out = ssh.send_recv_session("ls")
 |      print( ssh.send_recv_session("python") )
 |      print( ssh.send_recv_session("print('3')") )
 |      print( ssh.send_recv_session("import sys\nsys.executable") )
 |      print( ssh.send_recv_session("sys.exit()") )
 |      print( ssh.send_recv_session(None) )
 |      ssh.close_session()
 |      ssh.close()
 |      @endcode
 |
 |      The notebook :ref:`exampleofsshclientcommunicationrst` illustrates
 |      the output of these instructions.
 |
 |      @endexample
 |
 |  pig_submit(self, pig_file, dependencies=None, params=None,
redirection='redirection', local=False, stop_on_failure=False, check=False,
no_exception=True, fLOG=<function noLOG at 0x0000000008160E18>)
 |      submits a PIG script, it first upload the script
 |      to the default folder and submit it
 |
 |      @param      pig_file        pig script (local)
 |      @param      dependencies    others files to upload (still in the default
folder)
 |      @param      params          parameters to send to the job
 |      @param      redirection     string empty or not
 |      @param      local           local run or not (option `-x local
<https://cwiki.apache.org/confluence/display/PIG/PigTutorial>`_)  (in that case,
redirection will be empty)
 |      @param      stop_on_failure if True, add option ``-stop_on_failure`` on
the command line
 |      @param      check           if True, add option ``-check`` (in that
case, redirection will be empty)
 |      @param      no_exception    sent to @see me execute_command
 |      @param      fLOG            logging function
 |      @return                     out, err from @see me execute_command
 |
 |      If *redirection* is not empty, the job is submitted but
 |      the function returns after the standard output and error were
 |      redirected to ``redirection.out`` and ``redirection.err``.
 |
 |      The first file will contain the results of commands
 |      `DESCRIBE <http://pig.apache.org/docs/r0.14.0/test.html#describe>`_
```

```
 |      `DUMP <http://pig.apache.org/docs/r0.14.0/test.html#dump>`_,
 |      `EXPLAIN <http://pig.apache.org/docs/r0.14.0/test.html#explain>`_.
 |      The standard error receives logs and exceptions.
 |
 |      The function executes the command line::
 |
 |          pig  -execute -f <filename>
 |
 |      With redirection::
 |
 |          pig -execute -f <filename> 2> redirection.err 1> redirection.out &
 |
 |      .. versionadded:: 1.1
 |
 |  send_recv_session(self, fillin)
 |      Send something through a session,
 |      the function is supposed to return when the execute of the given command
is done,
 |      but this is quite difficult to detect without knowing what exactly was
send.
 |
 |      So we add a timeout just to tell the function it has to return even if
nothing
 |      tells the command has finished. It fillin is None, the function will
just
 |      listen to the output.
 |
 |      @param      fillin          sent to stdin
 |      @return                     stdout
 |
 |      The output contains
 |      `escape codes <http://ascii-table.com/ansi-escape-sequences-
vt-100.php>`_.
 |      They can be converted to plain text or HTML
 |      by using the module `ansiconv <http://pythonhosted.org/ansiconv/>`_
 |      and `ansi2html <https://github.com/ralphbean/ansi2html/>`_.
 |      This can be specified when opening the session.
 |
 |  upload(self, localpath, remotepath)
 |      upload a file to the remote machine (not on the cluster)
 |
 |      @param      localpath     local file (or a list of files)
 |      @param      remotepath    remote file
 |
 |      .. versionchanged:: 1.1
 |          it can upload multiple files if localpath is a list
 |
 |  upload_cluster(self, localpath, remotepath)
 |      the function directly uploads the file to the cluster, it first goes
 |      to the bridge, uploads it to the cluster and deletes it from the bridge
 |
 |      @param  localpath       local filename (or list of files)
 |      @param  remotepath      path to the cluster
 |      @return                 filename
```

```
 |
 |      .. versionadded:: 1.1
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  build_command_line_parameters(params)
 |      builds a string for ``pig`` based on the parameters in params
 |
 |      @param      params      dictionary
 |      @return                 string
 |
 |      .. versionadded:: 1.1
 |
 |  parse_lsout(out, local_schema=True)
 |      parses the output of a command ls
 |
 |      @param  out             output
 |      @param  local_schema    schema for the bridge or the cluster (False)
 |      @return                 DataFrame
 |
 |      .. versionadded:: 1.1
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

We open the connection using the variables stored two cells above:

[5]:
```
%remote_open
```

[5]:
```
<pyenbc.remote.ssh_remote_connection.ASSHClient at 0xa854fd0>
```

We try a simple command:

[6]:
```
%remote_cmd ls .
```

[6]:
```
<IPython.core.display.HTML object>
```

We then execute a python program on the remote machine, we first same the following code as a program:

[7]:
```
%%PYTHON exemple.py

import sys
print("path to python", sys.executable)
print("version ", sys.version_info)
```

The next command stores it as a file, uploads it to the remote machine and then executes it:

[8]:
```
%remote_py exemple.py
```

[8]: <IPython.core.display.HTML object>

We check it is different from the local version:

[9]:
```
import sys
sys.executable, sys.version_info
```

[9]:
```
('c:\\python34_x64\\python.exe',
 sys.version_info(major=3, minor=4, micro=3, releaselevel='final', serial=0))
```

If you want to use a different version of the interpreter, you can try (not available here):

[10]:
```
%remote_py -i=anaconda3/bin/python3.4 exemple.py
```

[10]: <IPython.core.display.HTML object>

Sometimes, you need an interactive shell such as Putty. Let's see how it works:

[11]:
```
%open_remote_shell
```

[11]: True

[12]:
```
%%shell_remote
ls
```

[12]: <IPython.core.display.HTML object>

[13]:
```
%shell_remote python
```

[13]: <IPython.core.display.HTML object>

[14]:
```
%%shell_remote

import sys
sys.executable
```

[14]: <IPython.core.display.HTML object>

To close the shell, we can just type:

[15]:
```
%close_remote_shell
```

[15]: True

And to close the connection:

[16]:
```
%remote_close
```

[16]: True

END

[17]:
```

```