Remote Debugging with eric7

date: 2023-01-10

Copyright Page

Disclaimer
The information in this document is subject to change without notice. The author and publisher have made their best efforts about the contents of this book, nevertheless the author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any loss or damages of any kind caused or alleged to be caused directly or indirectly from this document.

Remote Debugging with Eric7

This is written from a Linux perspective, but in principle it is valid for any operating system.

The Setup:
You need to develop and debug an application on a remote system and it is not practical or possible for you to just sit next to that computer and access it with a local keyboard and screen.

The inconvenience:
Let’s assume you only have access to the remote machine via ssh. Running an X-application, like an integrated development environment, is VERY SLOW – it may take many seconds for the graphical user interface to reflect what you just typed on the keyboard. This extreme sluggish behaviour seriously degrades your productivity and makes you scream in frustration. However, don’t blame ssh. In principle ssh is very fast and efficient, it just was not designed for such a use scenario. It is the wrong tool for your task.

A convenient solution – but not necessarily the optimum solution:
The installation of a remote desktop application, like NoMachine or ThinLinc, might be an appealing solution for some users. If you have a very nice and considerate network and system admin, then he or she may install this for you and you connect to the remote machine in all its glory as if you were there and with very little delay for most applications. If you are debugging some code, you will mostly likely not notice any sluggishness. However, your network and system admin has to consider many aspects that you as a user may be unaware of, like increased network traffic caused, security considerations etc so he or she may decide against such an option and there will be little you can do against that. And they might actually be doing you a favour by not installing it – well, in some cases.

Using the remote debugging capabilities of your IDE:
This leaves you with the option to take advantage of the remote debugging capabilities of your IDE in case it has such feature and in case this is documented in a way that an average user who is not a networking IT guru of the innermost circle can get this to work for him or her. I don’t belong to the *guru* group and it took me some googling and direct communication with Eric’s main developer Detlev, who is a very knowledgeable and helpful person, to get this to work for me and the deal with Detlev was that I would write a HowTo for anyone interested in doing the same. So here it goes.

In my scenario the remote computer is a virtual machine on a BIG computer hosting a HUGE data portal. This is why I want my code to run on that remote machine, because I don’t want to download Terrabytes or Petabytes of data that my code needs to play with. My remote machine is called *Skuggfaxe*[1].

Prerequisites:

1. You may need a VPN (to the network in which the remote system is living), to be active in order to be able to access your remote machine. If you do, fire it up. If you don’t know nothing about any VPN, try without and if you fail to open a SSH tunnel (see below), talk to your friendly network admin to get some clues.
2. SSH access to the remote system without being prompted for a password. The obvious choice here being a SSH key pair without setting a password for it. Google this, if you are not familiar with it. There may be other alternatives though.

[1] The name of Gandalf’s (LOTR) noble steed in its Swedish translation

3. It is also recommendable, though perhaps not strictly necessary (untested), that your ssh connection does not spit out messages. So keep your ~/.bashrc neat and clean or start with a very basic one. Once you get it to work, you can play around and add things that you like to add for personal choice or convenience.
4. You need to keep the source code on your local system and on the remote system in sync. At all times. You could do this by running an rsync-over-ssh command every time you make a change to the source code, but that is a very boring and repetitive task that you are likely to forget to do every now and then. I used the *mutagen* software (https://mutagen.io/) to do this for me. It is like a fancy rsync over ssh. I have set it up in such a way that my local system is the master copy and the remote system mirrors whatever I have on the local system. This seemed to be the best matching option for my needs, but you may choose differently.
5. Though you can have the source code on the remote and on the local system in different paths, it is advisable not to do so. Eric7 allows you to take care of this, but it is so much easier for my brain having the source code in the same paths on both machines… Alas, my SysAdmin in his humorous nature, created my account on the remote system with a different user name, though I had specifically requested to have the same user name (I really didn't need a 7th one…), but he wanted to give me a *nice* name. So no chance of having the same absolute paths. But with "*ln -s*" to the rescue I created a */home/RemotUserName* directory on my local system and mapped it in such way that the same absolute path as on the remote system resulted. This way even the Eric7 project and session files can be the same on either system. Most convenient.

Eric7:

Once you have the above prerequisites in place we can start worrying about remote debugging with Eric. Go to https://eric-ide.python-projects.org/eric-download.html and get the latest version and install it as usual. This needs to be a version >22.12.0 or else you won't be able to set the debug listening port. Then proceed….

1. Establish your automatic source code synchronisation, e.g.
   ```
   mutagen sync create -m one-way-replica -n PATH-TO-SOURCE-ON-A-
   (LOCAL-SYSTEM)  REMOTEHOST:PATH-TO-SOURCE-ON-B-(REMOTE-SYSTEM)
   ```

   ```
   (e.g.
   mutagen sync create -m one-way-replica -n skuggfaxe
   /home/arndt/dev/lumia/lumiaDA/lumia/
   skuggfaxe:/home/arndt/dev/lumia/lumiaDA/lumia/)
   ```
2. Mutagen is surprisingly persistent. Unless killed explicitly, it will survive a re-boot of your system without questions asked. You can check the status with:
   ```
   mutagen sync list
   ```
   The latter is also sufficient to revive the mutagen link, e.g. after a temporal VPN drop-out.

3. Next you establish an SSH tunnel for the ports the local Eric7 IDE and the remote debugger are communicating on. If the remote system were on the same local network as you with all ports open, you could skip this step and leave communication to the default ports. However, in most scenarios the remote system is on a remote or on a virtual network or both – as is the case for me. None of the TCP/IP or UDP ports on the remote virtual machine are exposed and are therefore not accessible directly from my local machine. That is also why I cannot install a solution like nomachine as a user, because the remote physical host that hosts the remote virtual system adds another abstraction layer and my SSH connection does not use the standard port 22 (the latter you configure in the *~/.ssh/known_hosts* file). Thus we need to funnel the port on which the remote debugger communicates on the remote virtual machine to a local port on which the Eric7 IDE "listens" (and talks back) with the following command (PORT4REMOTEDEBUGGER defaults to 35000 but can be anything – I chose to have the same port number in both directions eliminating potential confusion)

```
ssh REMOTESYSTEM -R PORT4IDE:localhost:PORT4REMOTEDEBUGGER
```

e.g. *ssh skuggfaxe -R 42424:localhost:42424*

This command tele-transports you onto the remote system (you should notice so on the command prompt). Beware that with this command in that terminal “localhost” means actually your remote system, which through the eyes of the remote debugger is its localhost. PORT4IDE on the other hand, denoted with the -R option for “remote”, is your local computer on which the Eric7 IDE will run. Port 42424 is the default port for this task for the Eric7 IDE, but (at the time of writing this) you do have to specify it in the IDE explicitly.

4. Fire up Eric7 on your local computer and load your project as usual.
   - Go to Settings → Preferences → Debugger → General *(see screenshot on the next page)*
   - Enable “Static Server Port” and set it to the value you used for PORT4IDE above. In my case I chose the default, which happens to be 42424
   - The “Remote Debugger” and “Perform Path Translation” options you can either specify here or on a per-project-level in the project settings. No need to set it in both locations. Whether you set these settings here or in the projects settings is a question of work flows and personal choices; if you want to tweak this on the project level or if you can get away with one general setting – your choice. I chose the project level in this example.
   - Click “apply” and “Ok” and close the dialog.
   - Under Project → Debugger → Debugger Properties (see also the screenshots below)
   - Set “Debug Client” to the remote debugger app as you would type it on the remote system. For me that is */opt/conda/envs/lumia/lib/python3.9/site-packages/eric7/DebugClients/Python/DebugClient.py*
   - Then enable “Remote Debugger” with “Remote Host” specifying the remote system name as you would hand it to any SSH command. For me that is simply “*skuggfaxe*” and the command or protocol to establish remote execution is “*ssh*”.
   - If remote and local paths to your source code are different, you have to enable “Perform Path Translation” – and if they are the same it does not hurt to enable it – and enter the absolute path to your source code tree respectively.
   - Click on “Ok”. This project setting overrides the general settings. I’m also attaching 2 screen shots to illustrate these 2 steps

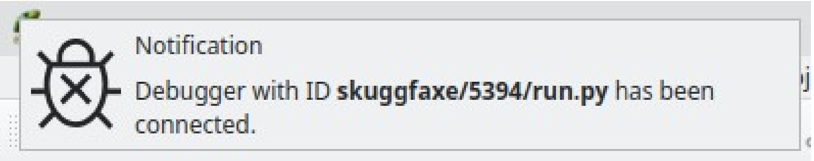
- Return to the terminal where you established the SSH tunnel, change into the directory where you would normally run the code you want to debug and start the remote debugger:

  ```
  python PATHTOYOUR/site-packages/eric7DebugClients/Python/DebugClient.py -p 42424 -w
  YOURWORKINGDIRECTORY -- YOURPATH/YOURCODE.py
  COMMANDLINE_OPTIONS_4YOUR_CODE
  ```

  Note the two minus signs before your script name. Example:

  ```
  python /opt/conda/envs/lumia/lib/python3.9/site-packages/eric7DebugClients/Python/
  DebugClient.py -p 42424 -w /home/arndt/nateko/data/icos/DICE --
  /home/arndt/dev/lumia/lumiaDA/lumia/run.py --optimize --start 20180101 --end 20180201 --rcf
  ./coco2_co2_v4.rc
  ```

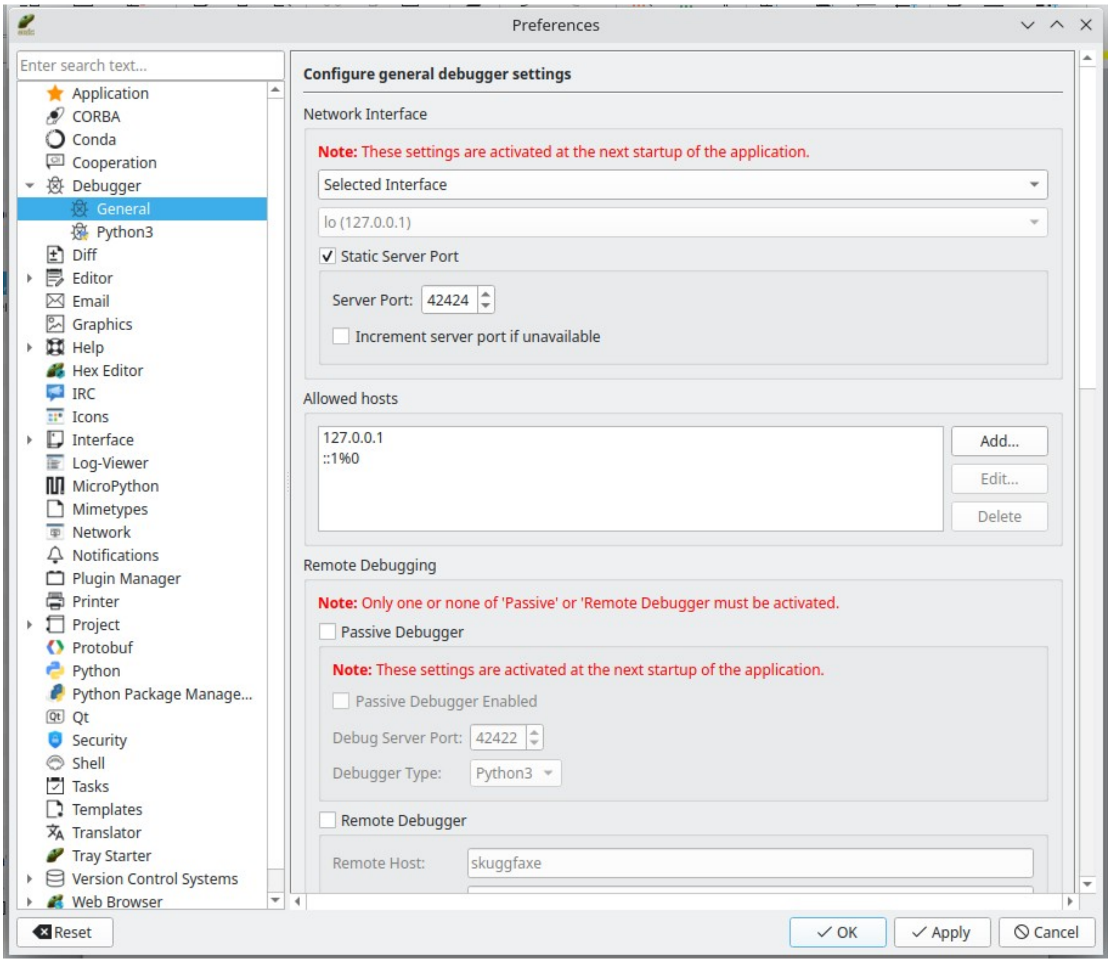
  - This should take your focus straight into the Eric IDE on your local system and you should get a brief confirmation message that the remote debugger has connect successfully with your IDE. Below is an example of such a message where the process number and script name may of course vary from yours.



Notification

Debugger with ID **skuggfaxe/5394/run.py** has been connected.

All things fair, any changes you make to your code locally will be pushed through to the remote system by mutagen, where the debugger runs the remote copy of your code on the remote machine. All breakpoints and inspection of variables should work as usual with the local debugger.



Configure general debugger settings

Network Interface

Note: These settings are activated at the next startup of the application.

Selected Interface

lo (127.0.0.1)

☑ Static Server Port

Server Port: 42424

□ Increment server port if unavailable

Allowed hosts

127.0.0.1
::1%0

Remote Debugging

Note: Only one or none of 'Passive' or 'Remote Debugger must be activated.

□ Passive Debugger

Note: These settings are activated at the next startup of the application.

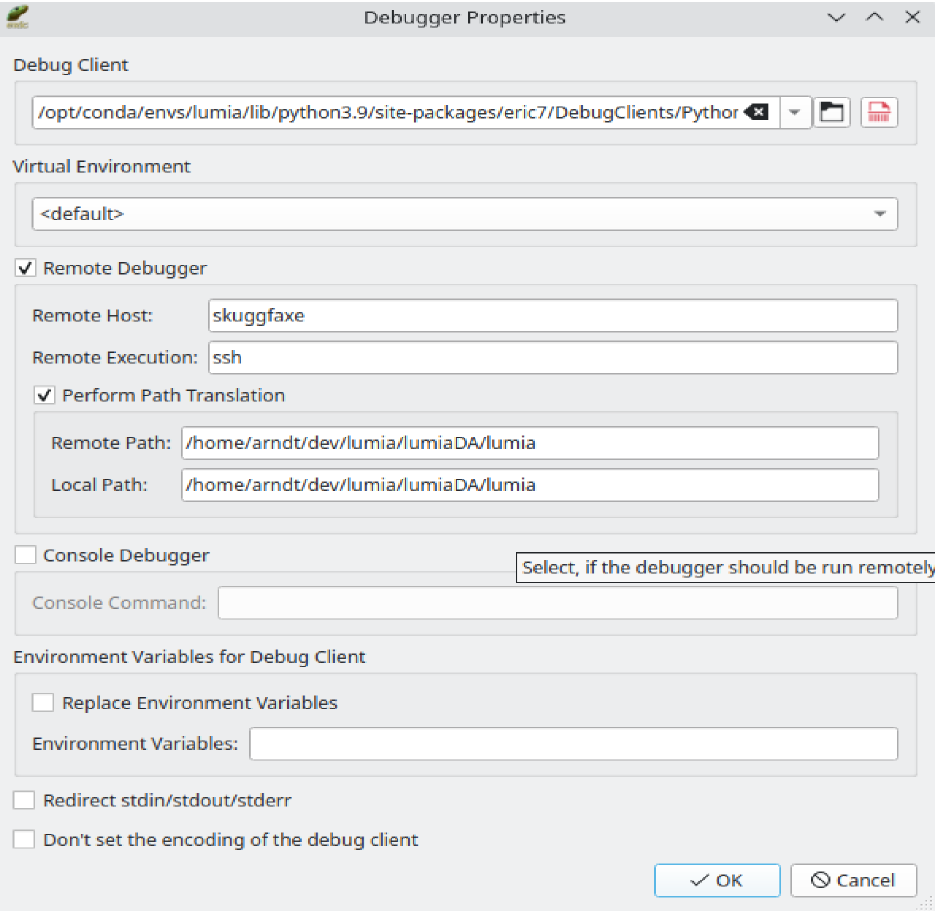
□ Passive Debugger Enabled

Debug Server Port: 42422

Debugger Type: Python3

□ Remote Debugger

Remote Host: skuggfaxe



Debug Client

/opt/conda/envs/lumia/lib/python3.9/site-packages/eric7/DebugClients/Pythor

Virtual Environment

<default>

☑ Remote Debugger

Remote Host: skuggfaxe

Remote Execution: ssh

☑ Perform Path Translation

Remote Path: /home/arndt/dev/lumia/lumiaDA/lumia

Local Path: /home/arndt/dev/lumia/lumiaDA/lumia

□ Console Debugger

Select, if the debugger should be run remotely

Console Command:

Environment Variables for Debug Client

□ Replace Environment Variables

Environment Variables:

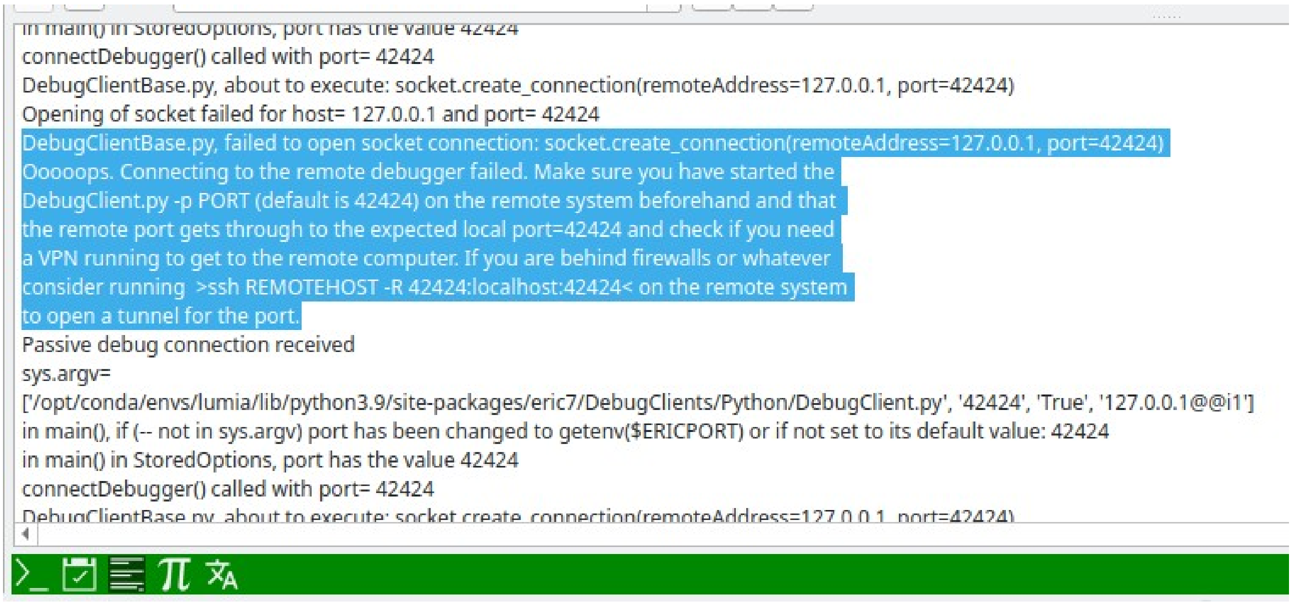
□ Redirect stdin/stdout/stderr

□ Don't set the encoding of the debug client

A last side note:

If things don’t work for you... ...be it first time or be it while working or going for a coffee and there is a network disruption, then first thing you should check out the Log-Viewer:

Should your VPN, internet connection etc. be disrupted, then Eric7 will actually give you a nice and helpful error message in the Log-Viewer window (see screenshot from when my VPN dropped out. The event is highlighted in blue. - Just reconnect the VPN, re-open the tunnel, restart the debugger on the remote system and continue.)



```
in main() in StoredOptions, port has the value 42424
connectDebugger() called with port= 42424
DebugClientBase.py, about to execute: socket.create_connection(remoteAddress=127.0.0.1, port=42424)
Opening of socket failed for host= 127.0.0.1 and port= 42424
DebugClientBase.py, failed to open socket connection: socket.create_connection(remoteAddress=127.0.0.1, port=42424)
Ooooops. Connecting to the remote debugger failed. Make sure you have started the
DebugClient.py -p PORT (default is 42424) on the remote system beforehand and that
the remote port gets through to the expected local port=42424 and check if you need
a VPN running to get to the remote computer. If you are behind firewalls or whatever
consider running  >ssh REMOTEHOST -R 42424:localhost:42424< on the remote system
to open a tunnel for the port.
Passive debug connection received
sys.argv=
['/opt/conda/envs/lumia/lib/python3.9/site-packages/eric7/DebugClients/Python/DebugClient.py', '42424', 'True', '127.0.0.1@@i1']
in main(), if (-- not in sys.argv) port has been changed to getenv($ERICPORT) or if not set to its default value: 42424
in main() in StoredOptions, port has the value 42424
connectDebugger() called with port= 42424
DebugClientBase.py, about to execute: socket.create_connection(remoteAddress=127.0.0.1, port=42424)
```

Have fun!