

Distributor

Version 1.5

User Guide

Frank Sonnenburg
Marcus Gastreich
Holger Claußen



© 2004 BioSolveIT GmbH
An der Ziegelei 75, 53757 St. Augustin, Germany
Phone ++49-2241-2525-0, support@biosolveit.de

Contents

1	Introduction	3
1.1	What's the purpose of the <i>Distributor</i> ?	3
1.2	Terminology	3
1.3	A typical Workflow	4
2	Installation	6
2.1	Requirements	6
2.2	Installation of the <i>Distributor</i> step by step	6
3	A Tutorial Introduction with Examples	8
3.1	Basic Usage	8
3.2	Getting Started	8
3.3	Following the Workflow Rationale	9
3.3.1	Configuration	9
3.3.2	Splitting the input files	10
3.3.3	Submission to the BQS	10
3.3.4	Merging and Output	11
3.4	Using <i>Distributor</i> from Python	11
3.5	Callback mechanism	11
3.5.1	Commandline Interface	12
3.5.2	Python Interface	12
4	Configuration	14
4.1	Design of Configuration Files	14
4.2	Hierarchy of Configuration Instances	15
4.3	Hierarchy of Configuration Sections	16
4.4	Option Substitution	16
4.5	User Arguments	16
5	Writing File Splitter Functions	18
5.1	A Minimum Template for <code>SplitMyType.py</code>	19
5.2	Defining New Options	20
5.3	Things to Ponder When Implementing <code>do_test_file</code>	20
5.4	Things to Ponder When Implementing <code>do_split</code>	21
5.5	Unittests for each <code>FileSplitter</code>	21
6	User Reference	22
6.1	<i>Distributor</i> commands	22
6.1.1	Kill <order dir>	22
6.1.2	Mailstats <order dir>	22

6.1.3	Merge <order dir>	22
6.1.4	Remove <order dir>	23
6.1.5	Resume <order dir>	23
6.1.6	Statistics <order dir>	23
6.1.7	Submit <nof jobs>	24
6.2	Options Reference	24
6.2.1	<i>Distributor</i> Global Options	25
6.2.2	FileType Options	26
6.2.3	Tool Options	27
6.2.4	BQS Options	27

Introduction

1.1 What's the purpose of the *Distributor*?

In short: The *Distributor* serves as a multi-purpose pre- and post-processing tool for distributed computing. Its strength lies in scenarios with large input files. The *Distributor* is independent from the compute tool itself. Any tool exhibiting a setup like

```
TOOL ← input_file(s) → output_file(s),
```

is suitable. Because this is standard UNIX behaviour, a multitude of tools can be used with the *Distributor*, amongst them all BioSolveIT cheminformatics tools, such as *FlexX*, *FlexS* and *FTrees*.

In molecular virtual screening, one usually deals with several thousands of compounds, or their respective descriptor files. On today's compute farms, an efficient, yet simple parallelisation can be achieved by simple chopping up the input file(s) and make the chemistry program compute the much smaller chunks in parallel.

The basic idea behind the *Distributor* now is to have a tool which frees the user from the burden of manually cutting large jobs into pieces, and distributing them across, say, a Linux cluster. Having arrived there, one certainly wants to profit from additional features, such as an inherent load balancing, automated error reporting (for example, through email) boundary conditions on scheduling, and much more.

The *Distributor* achieves these goals by being based on maximum generic approach: We have implemented a system which goes hand in hand with the (often pre-installed) queuing system (here: BQS). Thus, most of the latter challenges are taken care of by the queuing system. Triggering its events, preparing, and re-assembling the output data from distributed compute sources, that's what the *Distributor* is about.

The *Distributor* has been implemented with Python, an open, free, all-purpose programming and scripting language. Interfacing the *Distributor* to other tools is thus facilitated for future requirements.

1.2 Terminology

There are a couple of technical terms which we would like to clarify before continuing with this manual to avoid confusion. These are:

- *Item* – Smallest block of input data.
- *Segment* – Set of items in one file.
- *Input file* – File that can be processed by a tool. An input file consists of one or more segments. A *set* of input files consists of several input files of the same type.

- *Single Job* – Tool computation on a single item.
- *Job* – Computation on a segment.
- *Order* – Computation on an input file or a set of input files, which is/are divided into segments.
- *Batch queuing system* – A program for spreading a big number of computations over a computer pool in order to balance the workload fairly between these machines.
- *Tool* – An arbitrary computer program used for the actual computation of the input data. The *Distributor* will cut the input files into segments and “feed” these to multiple instances of the tool.
- *FileSplitter* – A *Distributor* module, that can handle, i.e. cut, a special type of input files.

1.3 A typical Workflow

The typical workflow is demonstrated in the figure beneath (Fig. 1.1) It may look complicated, but that’s only due to the fact that there are many copies of the same line of computation.

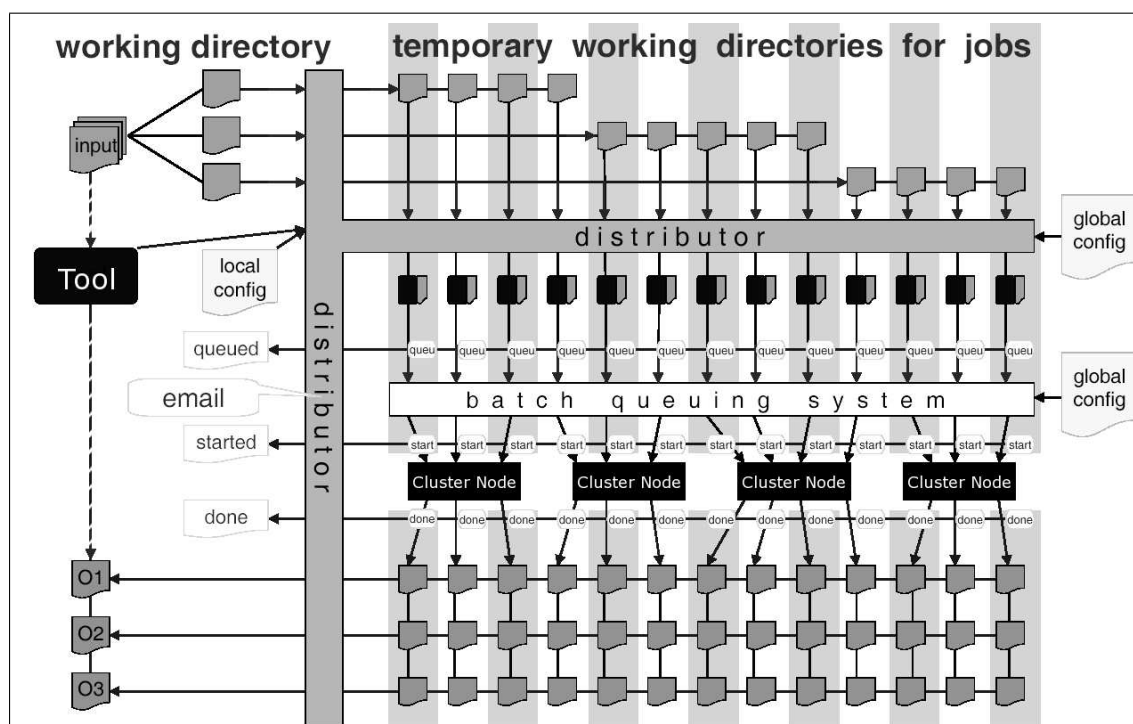




Figure 1.1: The basic workflow when working with the *Distributor*.

Let us start in the upper left of the figure and read it in clockwise order; “Tool” stands for any tool meeting the above-mentioned requirements, for example the *FlexX* docking suite of programs.

1. The input which would usually be directly processed by the Tool (dashed lines in the left area) has to be split into pieces. This is meant to be visualised by the vertical block “distributor” - which we call “the Splitter”. Splitting rules are pre-defined for some tools, or input file types, e.g., mol2 files and feature tree files, but other file types can be defined just the same. There is one decisive consequence associated with this concept: The user loses an instance counter across the entire, original input file. So, be advised not to use any counters in your scripts which rely on the sequence of instances in your input file; for example: do not use a counter for compounds 1–50 in your input file to recognise them as the active compounds during a screening. 
2. Every chunk, the so-called “segment file” receives its own work space, i.e., an own directory, so that no mixing up of instances can occur. This storage is temporary; if a job crashes though, they will still be available and have to be deleted manually.
3. The *Distributor* now assembles work packages consisting of a tool call and a segment file. This is displayed just underneath the horizontal “distributor” block. The assembly is controlled by a global distributor configuration file which may be overridden by a local configuration file, e.g., in the user’s home directory.
4. Subsequently the job packages are queued for computation and given over to the batch queuing system BQS. After being queued, a notification may be issued to the scientist.
5. Once started, the user is informed, and the queuing system covers for load balancing of the duties. Email notification to the user can be evoked if wished. Depending on configuration and the size of the segment-files, a processor may get several instances for computation.
6. Once the work is done, the user will be informed and the *Distributor* compiles all output files to the respective output files which would otherwise have been assembled sequentially (O1 to O3 in our example).

Please remark, that the item order in the input files generally is *not* carried over to the output files! But the *Distributor* assures, that if the output of job Z comes behind the output of job X in output file A, then output file B will show the same order. In other words, the order of job output corresponds between each output files, but not between output and input. 

Installation

2.1 Requirements

The *Distributor* relies on python technology. Python is an open-source programming and scripting language free to the public. Because of its rapid development, we recommend to use a current version. Here are the current minimum requirements for the *Distributor*.

- Python 2.3 or higher.
- Batch queuing system: OpenPBS or others on demand.

The batch queuing system is assumed to run stable and to be ready to accept orders. Try out a simple test before continuing; this little script:

```
% echo 'echo Hello, World!' > test.sh
% qsub test.sh
```

should run without problems. Assuming you have OpenPBS, then, after having executed the script, there should be a file named `test.sh.o<jobID>` containing the output «Hello, World!».

If this fails, please consult your systems administrator and present any error messages you may have encountered during this test.

2.2 Installation of the *Distributor* step by step

- Unpack `BSI_distributor-<ver>.tgz`.
- Move the directory `distributor` and `BSI_distributor.pyc` to a location included in `$PYTHONPATH`, e.g., `/usr/lib/python2.3/site-packages`. Alternatively, insert location of the distributor directory into `$PYTHONPATH`.
- Copy (and edit) `BSI_distributor.cfg` to one or more of the files (for more details see ch. 4):
 - `/etc/BSI_distributor.cfg`
 - `$HOME/.BSI_distributor.cfg`
 - `<working dir>/BSI_distributor.cfg`
- Move `BSI_distributor.py` to a location included in `$PATH` (or insert the location of `BSI_distributor.py` into `$PATH`).

- Make sure that the file `BSI_distributor.py` is executable (`chmod a+x BSI_distributor.py`).

A Tutorial Introduction with Examples

3.1 Basic Usage

In principle, there are two ways to start the *Distributor*. Either you start it as a “stand-alone” program, or you import it as a so-called python module into an already called python program. Here is how it goes (line breaks here are for reasons of limited paper size. The % and >>>- signs signify the UNIX or python prompts, respectively):

- Calling From the console as a stand-alone program:

```
% BSI_distributor.py [<option> ...] file [file ...]
```

In case the standard python you use is version 2.2 or lower, please try:

```
% python2.3 <install dir>/BSI_distributor.pyc [<option> ...]  
file [<file> ...]
```

or

```
% python2.3 'which BSI_distributor.py' [<option> ...]  
file [<file> ...]
```

- From within python:

```
>>> import distributor  
>>> orderID = distributor.main(<file>, ...,  
                             <opt_with_underscores>=<val>, ...)
```

If you do not encounter any warnings or error messages, the *Distributor* has properly been called.

3.2 Getting Started

Start the command line help of the *Distributor* by entering `BSI_distributor.py -h` at the command line.

This will print the usage information and all available options to the screen (online help). Arguments may be given to the *Distributor* on the command line separated by blanks. “Options” are special arguments that start with one or two minus signs. Most of them again take an argument which has to be given after another separating blank. Alternatives for option arguments are obtainable from the online help as well. Arguments have

to be given in quotes if they contain spaces or special shell characters. Any “long option” (that is, the ones which start with two dashes “--” may be abbreviated as long as the prefix is unique, e.g., `--statistics` can be accessed by `--stat` as well.

Examples:

```
% BSI_distributor.py ligands.mol2
% BSI_distributor.py -F sdf a.sdf b.sdf
% BSI_distributor.py -F AsciiPattern --pattern MOL
  --tool "grep -c Water" c.molecule
```

To get the unique order directory without parsing the *Distributor* output, use `--print-dir` or `-D`. This will print only the order directory to stdout, so you can use the following shell command to store the order directory into an environment variable:

```
# sh, bash
% export order_dir=`BSI_distributor.py ligs.mol2 --print-dir`
# csh, tcsh
% setenv order_dir `BSI_distributor.py ligs.mol2 --print-dir`
```

3.3 Following the Workflow Rationale

A very rough overview of *Distributor*'s workflow looks like this:

1. Configuration
2. Splitting the input files
3. Submission of jobs to the batch queuing system (BQS)
4. Merging (in the background) and output

We will try to stick to these points to help you read this section.

3.3.1 Configuration

In order to start a computation one or more input files are needed. *Distributor* first will read

- `/etc/BSI_distributor.cfg`,
- `$HOME/.BSI_distributor.cfg` (note the leading dot), and
- `<working dir>/BSI_distributor.cfg`,

in the above order. Additionally you might specify another config file via command line option `--config` or `-c`. Finally, the *Distributor* looks for options given on the command line. More recent option values override previously read ones.

See also *Configuration* (ch. 4 on page 14) for more details.

3.3.2 Splitting the input files

After having configured the current session, *Distributor* splits the given input files into segment files and stores them into temporary directories, which may take a couple of minutes for large files (about 1 minute for 50 MByte with a one-pass file splitter).

3.3.3 Submission to the BQS

The third step is creating and submitting jobs to the batch queuing system. *Distributor* will print the order directory on the screen in a message such as:


```
>> Splitting input files ...
>> 4 job(s) to submit, order-dir = /home/.../BSI_theta_20040420_105733_010645
....
```

The “order id” which is the last path component of the “order dir”, consists of a BioSolveIT-specific prefix, the current computer’s name, the date, time, and the process ID of the initiating *Distributor* call.

For each given input file, a separate subdirectory <order dir>/<input dir> is created, and each job will be executed in a separate <job dir> under <input dir>, which contains the segment of the input file for this job, the job script, and additional status files.

Now, let us assume your order is running. At any time, you can ask for the actual status of this order:

```
% BSI_distributor.py --statistics <order dir>
```

Note: For arguments on special *Distributor* commands like “--statistics”, see the corresponding reference section on page 22. 

The output of *--statistics* looks like

```
BioSolveIT Distributor Status Information
=====

Order ID:                BSI_theta_20040617_142516_007203
Order Directory:         /home/frank/tmp/BSI_theta_20040617_142516_007203
Output Directory:       /home/frank/tmp/BSI_theta_20040617_142516_007203/BSI_output
-----
Input files:             /home/frank/project/distributor/src/bionet.mol2
-----
Job count:               5
BQS job ids:             3766-3770
-----
Job id's done:          -
Job id's merged:        -
Job id's removed:       -
Job id's failed:        -
-----
Done items per job
-----
00001: 27  00002: 15  00003: 35  00004: 31  00005: 22
```

Note:
Entries ‘a-b’ are ranges ‘a, a+1, ..., b-1, b’.

You will find a description of this statistics output in section 6.1.6 on page 23.

3.3.4 Merging and Output

You should not really remark the merging of the jobs outputs, since it is performed in the background. Finally, output summary of your (hopefully successfully) executed job is stored in `<order dir>/BSI_output`. If they did anything, there will be two files, `stdout` and `stderr` which contain the standard output (what you would have seen on screen), and standard error output of the tool; usually, both would normally be printed to the screen. Additionally, you will find all other files produced by the tool *in its temporary* `<job dir>`.

If the size of an output file exceeds *max-file-size* (specified in bytes), the output file will be split into files `<fname>_part_<no>.<extension>`.

3.4 Using *Distributor* from Python

Since the *Distributor* is written entirely in Python, it's almost obvious to offer an interface to Python as well, additional to the above described command line interface.

To use the *Distributor* in your python script, just import the module "distributor", which provides a single function 'main'. Parameters to 'main' are:

- *input file(s)*: each file name given as a string parameter
- *options or commands*: option or command as named argument, where "-" (minus-sign) must be replaced by "_" (underscore). Argument values may be:
 - for options: strings, integer or float numbers or "True" for "flag-options" like `--print-dir`.
 - for commands: string with order directory (recommended) or order id.

Return value of 'main' is the order id or order directory if specified `--print-dir`. For command "statistics", `distributor.main()` returns a dictionary containing order information. See section 6.1.6 on page 23 for a reference on the content of this info dictionary.


Example:

```
import distributor
order_dir = distributor.main('lig/ldwd.mol2', 'lig/another.mol2',
                             segment_size=30, print_dir=True)
:
info_dict = distributor.main(statistics=order_dir)
```

3.5 Callback mechanism

A segment file consists of a bunch of items, the number of items is configurable with `--segment-size`. A job in terms of *Distributor* is a process or tool run on a compute node, which consumes the segment file sequentially item by item downwards. Each computation on an item is independent from any other item. When you are handling with large input files and even the segment files contain large numbers of items, the following *Distributor* feature might be very helpful:

If the tool fails on computing item x , the *Distributor* automatically restarts the same segment from item $x+1$ onwards.

Requirement: The tool has to notify the *Distributor* after each successfully processed item.  This is done by calling a tiny python script called “BSI_ItemDone.py”, which is to be found in each job’s current working directory. Additionally, before processing the first item an initializing call has to be done.

You can either execute this script as a command line call (Command Line Interface) or import this piece of code as a python module (Python Interface) if your tool is a python script anyway. The following sections show examples for these two interfaces.

3.5.1 Commandline Interface

Do init call at the very beginning of your tool script:

```
% python BSI_ItemDone.py -i
```

Inside your loop over each item, call notifying script after having processed an item:

```
% python BSI_ItemDone.py
```

Example (FlexX batch script, “!” performs a system call):

```
# Initialize distributor callback mechanism
# -----
!python BSI_ItemDone.py -i

receptor                                # change into receptor menu
  read 4dfr                               # read protein
end                                        # leave receptor menu

# Loop over each item in segment file, given in $(input)
# -----
FOR_EACH $(mol_nr) FROMTO 1 $(mol_count)
  ligand                                  # change into ligand menu
    read $(input) $(mol_nr)             # read ligand to be docked
  end                                      # leave ligand menu

  docking                                 # change into docking menu
    complex all                          # build up complex
    ...                                   # ... some postprocessing and output ...

# Notify distributor of completion of current item
# -----
!python BSI_ItemDone.py

END_FOR
```

3.5.2 Python Interface

The callback script also provides a simple python interface with two module functions:

```
>>> from BSI_ItemDone import init_callback, distributor_callback
```

Example:

```
#!/usr/bin/python
import sys
from BSI_ItemDone import init_callback, distributor_callback

# Initialize distributor callback mechanism
# -----
```

```
init_callback()

# Loop over each item consisting of a single line
# -----
for line in file(sys.argv[1]):
    # Very silly, just add pi to number read from file
    print float(line) + 3.14159265

    # Notify distributor of completion of current item
    # -----
    distributor_callback()
```

Configuration

The *Distributor* is highly configurable. Configuration is essentially made through configuration files. Their design and contents are covered below.


4.1 Design of Configuration Files

Distributor configuration files are stored in the common so-called Windows INI format, which is supported by the Python standard module “ConfigParser”. Config files are divided into sections by section headers, e.g.

```
[Tool FlexX]
```

The section body consists of name/value entries, e.g.

```
flexx-script = /home/user/dock.bat
```

Note: Section titles are case sensitive (whereas option names are not). 

Each entry spans at least one line. Name and value are separated by the first equal sign (=). Whitespaces before and after a =-sign are ignored. Entries may span multiple lines, if the second and following lines are indented. Empty lines and comment lines (characters ‘#’ or ‘;’ in the first column) are ignored. Inline comments (i.e., comments behind an entry on the same line) will most probably cause nonsense: the comment will be added to the value.

The *Distributor*’s configuration files are split into four general sections:

- [Distributor]
- [BQS DEFAULT]
- [FileType DEFAULT]
- [Tool DEFAULT]

The last three have specialized sections with corresponding section title prefixes. For the current version of *Distributor* (which is 1.5), available special BQS sections for each supported batch queuing system are

- [BQS OpenPBS]
- [BQS SunGridEngine]

Each FileSplitter module, that comes with the *Distributor*, can be configured by a special *FileType* section; currently available are

- [FileType AsciiLine]
- [FileType AsciiPattern]
- [FileType ftree]
- [FileType mol2]
- [FileType sdf]
- [FileType sln]


- [FileType smiles]

Dealing with tools is even simpler: You can define any *Tool* section you like! For example:

- [Tool FlexX]
- [Tool MySecondChoiceDockingTool]

The main option to define in a tool section is simply *tool*. You have to specify the tool executable with absolute paths and options as needed. The segment filename then will be appended to this tool string separated by a blank as a command line argument. If you need to supply the segment filename in a different way, you can specify the *segment-placeholder* character (defaults to @) in the tool string, which will be replaced by the segment filename. In this case, the segment filename will not be appended as a command line argument. Example:

```
tool = /home/user/bin/flexx -b %(flexx-script)s -a '$(input)=@'
```

Note: The tool string is executed in a shell environment, so be careful to “escape” parts of the string, which are meaningful to the shell, by enclosing in apostrophes (') as done in the previous example with the trailing argument “-a '\$(...)=...’”, otherwise the shell would try to expand the string inside “\$()” as a shell variable. 

The vast majority of options is set underneath [Distributor], e.g. *mail* or *mail-interval*. Options related to FileSplitter are stored in [FileType DEFAULT] or a specialized FileType section variant, likewise for Tool and BQS. Please refer to the template BSI_distributor.cfg, to determine where to put which option.

To fully understand how the *Distributor* is configured, we will get into more detail with respect to the basic configuration concept implemented with the *Distributor*. There are two configuration hierarchies:

- Configuration instances, these are configuration files and command line options
- Configuration file sections

These two hierarchies are described in the following two sections of this user guide.

4.2 Hierarchy of Configuration Instances

Configuring a *Distributor* session is done in a maximum of six subsequent steps.

1. *Distributor* first starts with a set of defaults set by the programmer.
2. Read /etc/BSI_distributor.cfg (if available)
3. Read \$HOME/.BSI_distributor.cfg (if available)
4. Read ./BSI_distributor.cfg (if available)
5. Read special config file, if specified at command line via *--config*
6. Parse command line arguments

Values in later stages override previously defined ones. A missing configuration file will *not* lead to an error. Also, each config file may specify only a subset of options.

The effective configuration setting used with a specific order is stored under <order-dir>/BSI_distributor.cfg. In case you get confused with multiple configuration files, you can check this file and assess, whether your final settings are the ones you expected. You can reuse this “compiled” or resulting configuration file, in which even your command line options of this order are stored, for subsequent *Distributor* orders by specifying the *--config* option at the command line.

4.3 Hierarchy of Configuration Sections

The configuration files are divided into several sections, as described above. The template `BSI_distributor.cfg` specifies where to put each option.


You may want to place options in other sections keeping in mind the side effects of doing so. There is an option lookup order in the config sections:

1. [Distributor] takes the highest precedence over any other section entry. Only entries in the same section *in following instances* may override a setting made here.
2. Special package sections, e.g. [Tool FlexX], are assigned at second level, i.e., only options not defined before are set now.

If you specify the same option in more than one active section, e.g. *segment-size* in [Tool FlexX] and [FileType mol2], the following order is used to resolve conflicts:

- (a) [Tool *]
 - (b) [FileType *]
 - (c) [BQS *]
3. Default package sections are assigned at latest, e.g. [Tool DEFAULT]. Settings made here are only used if they have not been found in any of the higher levels.

Please consider the following when defining options in a different section:

- When placing a special option, e.g., *tool* in [Distributor], the option value of *this* entry is used rather than the value specified in [Tool <tool-section>]! This “feature” comes quite handy when playing around with the effects of some options without bothering about the section structure.
- The reverse case: Setting a main option, e.g., *segment-size* only in [FileType ...] sections will enable you to choose the segment-size implicitly with the *filetype-section* option.
Please keep in mind: Once an option is set in the main section, *Distributor* will not check for this option in other sections. 

So, a good rule-of-thumb is: Rather define options in subsections than in the main section.

4.4 Option Substitution

Any option value can be substituted in other options, e.g.,

```
[Distributor]
segment-size = 30
[Tool Viewer]
tool = /usr/bin/viewer --mol-count %(segment-size)s
```

The value of an arbitrary option is inserted in another option’s value by using the option name surrounded by ‘%(’ and ‘)s’.


4.5 User Arguments

Any options with prefix “arg-” are recognized as user options or *user arguments*. The goal of user options is to render the configuration procedure even more flexible. Example:

```
# BSI_distributor.cfg
[Tool my]
tool = /usr/bin/my_tool -c %(arg-cfgpath)s/my_tool.cfg
arg-cfgpath = /home/user/config

# commandline
% BSI_distributor.py input.txt -T my -a cfgpath=/tmp
```

User arguments can be defined at the command line with “*--arg <name>=<value>*” or “*-a <name>=<value>*”. This will define “*arg-<name>*”. User options (as any other option) can be used in other options as python-style string placeholders with the format “*%(<opt-name>)s*”, e.g., “*%(arg-script)s*”


There is no default for user options! So, if used in another option as placeholder, you are urged to set a value for this placeholder — either in a config file or at the command line. 

Writing File Splitter Functions

Apart from several cases already provided with this distribution, the *Distributor* cannot “know” how to split input files, because this evidently depends on the nature of the input. For example, a molecule in a mol2 file is initiated by the Keyword @<TRIPOS>MOLECULE. For a feature trees file, the situation is different; here the keyword is @ftree. So, what the *Distributor* needs are certain rules according to which input files are split prior to computation. We will call these rules the “file splitter functions”.

Let us assume, such a file splitter function were currently not available for your special problem. Then, you will have to write one. But, never fear, writing one’s own FileSplitter modules is quite easy if you are a little familiar with python.

Here are the required steps along with an example:

1. Copy the template
`<install dir>/distributor/FileSplitter/SplitExample.py`
to a file named `Split<MyType>.py`.
2. Replace `<MyType>` by a speaking name for the type of files you’d like to split. This name will later be used in a special section as an option `--filetype-section/-T` (case  sensitive).
3. In your new file, find the line

```
class SplitExample(FileSplitter):
```

We will now substitute this so-called “class” `Split<MyType>` with your requirements. Your new “options” will go a bit below the class definition, in fact to where now the following lines are:

```
defaults = {  
    }
```

Ok, there are only two points which have to be covered:

- Enter the new options within the class dictionary ‘defaults’ as desired; if you do not succeed, have a look into `SplitAsciiLine.py`. For example, if we wanted to chop ASCII files in chunks of single lines, then we could do:

```

        defaults = {
            # One item consists of n lines, here: n=1
            "item-size": 1
        }

```

- Re-implement the so-called “methods” ‘do_test_file’ and ‘do_split’. They follow a bit further below in our example file and start with the lines

```

# -----
# Implementation of hook methods -- see FileSplitter/__init__.py
# -----

def do_test_file(self, filename):
    and
def do_split(self, start=1):

```

4. Save your file in the directory where `SplitExample.py` was, i.e., into `<install dir>/distributor/FileSplitter`.

We have entered a couple of comments and interface specification in `SplitExample.py`. If you are currently not aware of the location of this file or if you’d prefer a short summary, you can always simply enter the following commands (the first line from the following block calls python as we did before):

```

% python
>>> from distributor.FileSplitter import SplitExample
>>> help(SplitExample)

```

5.1 A Minimum Template for `SplitMyType.py`

From the above, you should have learned the structure of a file splitter function. To make writing a file splitter more convenient, we thought we should provide you with a template for the files. Similar to the `SplitExample.py`, here comes the respective minimum template, `SplitMyType.py`

```

"""
User defined FileSplitter module SplitMyType [more description ...]
"""
from __init__ import FileSplitter

class SplitMyType(FileSplitter):
    defaults = {}
    def do_test_file(self, filename):
        return True
    def do_split(self):
        input = open(self.filename, 'r')
        output = self.create_next_file()
        output.write(input.read())
        output.close()
        return True

```

The three first lines just give the usage information. the `from`-line says that we will import functions from other modules that are supplied with the *Distributor*. Next is the specification of the options (in `defaults`) (see next paragraph), followed by the two functions we already met before and for which more explanations will be given below.

5.2 Defining New Options

Maybe, you will want to give users the possibility to “fine tune” the file splitting progress; or, you definitely require the user to supply you with some information which you simply *a priori* do not know and have no defaults for. This is the scenario for inclusion of options. An example would be the number of lines which form an item when chopping up an ASCII file. As shortly said before, you need to define the call to options in the “class dictionary” ‘defaults’-section.

The concept behind the class dictionary is such that in the configuration file, the user will have a special keyword, called “key”, under which certain “options” are given. These options should have defaults in the *Distributor*. The “Keys” now are the above-mentioned “option names” (e.g., `pattern`, and “values” are the default values for these options, unless re-defined in the `config-file(s)` or at the command line, e.g., `\$\$\$\$\$`. For example, this is how the `defaults`-block in `Split<MyType>.py` may look like:

```
defaults = {
    # The pattern is a regular expression
    # (rules apply as in Python standard module 're')
    "pattern":          "\$\$\$\$\$",
    "no-keep-pattern":  False,
    "keep-pattern-before": False,
    "keep-pattern-after": False,
}
```

Now here comes one possibility for a configuration file, e.g., `BSI_distributor.cfg`. The hash marks denote comment lines. Our pattern will be `NEW_ITEM`. The remainders of this example are meant to clarify things a bit more.

```
[FileType AsciiPattern]

# (don't use quotes in the config.file)
pattern=NEW_ITEM

# pattern
# -----
# Default:      \$\$\$\$\$
# Description:  Pattern (regular expression with syntax of Python's standard
#              module re) to split items from each other.

# no-keep-pattern
# -----
# Default:      False
# Description:  Default is to keep pattern between items in a segment file. Use
#              this option to turn this behaviour off.
...
```

To access an option value in python code, use the method `self.get(optname[, default])`. If you are unsure whether a user argument, such as “`arg-subtype`”, is defined, it is advisable to supply a default for the value of this option; otherwise the *Distributor* will raise a `DistError` exception: “Option ‘xyz’ not defined”.

5.3 Things to Ponder When Implementing `do_test_file`

There is one argument besides `self` to `do_test_file`: it’s the filename. Check special requirements for splitting this file by your `FileSplitter`.

This method returns `True` on success, `False` otherwise.

5.4 Things to Ponder When Implementing `do_split`

As the filename suggests, the task of this method is to create segment files from input files. `do_split` is invoked once for each input file; its arguments are `self` and an integer number `start`. `start` is the number of the item (one-based) in the input file to regard as the starting item in the first segment. In other words: do ignore the first `start-1` items in the input file before splitting. This is necessary to implement the callback mechanism, see section 3.5 on page 11.

Currently available methods and instance members are:

- `self.filename` – The current file which is to be split
- `self.nof_items_in_segment` – The number of items per segment file (specified by user via `-segment-size/-s`)
- `self.get(optname)` – A function which accesses option values
- `self.create_next_file()` – Retrieve an empty file object which will be opened for writing and storing the contents of the next segment

This method returns `True` on success, `False` otherwise.

5.5 Unittests for each FileSplitter

We use the python standard library module “unittest” to perform simple tests for each implemented FileSplitter. If you are not familiar with the usage of unittest, please refer to the python online documentation. The tests are to be found in the source files of each FileSplitter module. The content of a dummy file is included in each test source. To run the test for a special FileSplitter module, say `Splitmol2.py`, either you have to execute this module like a python script from the command line:

```
% python <install dir>/distributor/FileSplitter/Splitmol2.py
```

or you may take this way:

```
>>> from distributor.FileSplitter import Splitmol2
>>> import unittest
>>> unittest.main(Splitmol2)
```

```
..
```

```
-----
Ran 2 tests in 0.034s
```

OK

User Reference

This reference shall serve as a quick guide and overview to the *Distributor*. We first access the commands the *Distributor* comes with, then we will provide an overview of all available options. For further reading on handling with option, please see the section *Configuration* in chapter 4 on page 14.

6.1 *Distributor* commands

Every *Distributor* command referring to an order you started needs a reference to this order; in fact, it's a parameter to the *Distributor*. To specify this parameter, you can either use the "order-ID", or, more recommended¹, the complete order directory including the order-ID as last path component. Example:

```
BSI_distributor.py --stat /home/frank/test/tmp/BSI_theta_20040514_124457_029818
```

6.1.1 Kill <order dir>

Delete or stop jobs in queues.

A bad scenario is the one when the BQS is currently down. Then, the *Distributor* cannot be aware of queued jobs. Evoking this command in such a situation will effect in running jobs be killed on the respective nodes by system calls. The distributor will connect to the nodes where jobs are running. In fact, this is achieved through the use of the secure shell `ssh` (which itself is configurable using the option `ssh`). So you will have to have proper permissions on the compute nodes. Once the batch queuing system is up again after the issue of a `kill` command, most systems re-schedule their jobs which have not been started before the system went down. To avoid this, there is currently no other way but to kill your order a second time after the BQS has been restarted.

6.1.2 Mailstats <order dir>

Initiate a cron job which sends status information via email every *mail-interval* minutes. This command is always invoked on startup of an order unless the option *mail-interval* is set to 0 (zero).


The output format is the same as from the command `statistics`, see below.

6.1.3 Merge <order dir>

Initiate a cron job which merges done jobs to global output files.

¹In case you used a different *tmp-dir* setting for this order, the *Distributor* wouldn't find this order if only the order-ID had been supplied.

6.1.4 Remove <order dir>

Stop the order if still running and remove the order directory. Beware! Any information  and output files will be lost after issuing `remove`! Global output files will be deleted, too, unless you have specified an output directory (option `output`), which is not a subdirectory of the order directory.

6.1.5 Resume <order dir>

Resume a previously started order.

6.1.6 Statistics <order dir>

Print statistics for the specified order in this format:


```
BioSolveIT Distributor Status Information
=====
Order ID:                BSI_theta_20040617_142516_007203
Order Directory:         /home/frank/tmp/BSI_theta_20040617_142516_007203
Output Directory:        /home/frank/tmp/BSI_theta_20040617_142516_007203/BSI_output
-----
Input files:             /home/frank/project/distributor/src/bionet.mol2
-----
Job count:               5
BQS job ids:             3766-3770
-----
Job id's done:           -
Job id's merged:         -
Job id's removed:        -
Job id's failed:         -
-----
Done items per job
-----
00001: 27  00002: 15  00003: 35  00004: 31  00005: 22
```

Note:

Entries 'a-b' are ranges 'a, a+1, ..., b-1, b'.

Let's have a look at the fields shown in the above sample statistics output.

- *Order ID* – Unique ID for current order consisting of: BioSolveIT prefix `BSI`, computer name where *Distributor* was started, date, time and *Distributor* process id.
- *Order Directory* – Temporary (master) directory, where *Distributor* creates temporary segment files and status information files.
- *Output Directory* – Path for merged output files.
- *Input files* – User supplied input files *plus* restarted segment files, if a job failed and tool supports callback mechanism; see also section 3.5 on page 11.
- *Job count* – Number of initially started jobs *plus* restarted jobs, see above.
- *BQS job ids* – The batch queuing system's job ids for this order. This entry (as well as the following four lines) is a *range* of numbers, i.e., in the above example, the following job ids are assigned: 3766, 3767, 3768, 3769 and 3770.

- *Job id's done* – The tool has processed this item, either successful or not.
- *Job id's merged* – The output of these jobs are merged to the global output files.
- *Job id's removed* – The segment has been processed successfully and the output has been merged successfully; so the temporary job directory has been removed.
- *Job id's failed* – These segments caused the tool to exit with an error status; nevertheless *Distributor* tries to merge the output. But the temporary job directory will not be removed, because user may want to further inspect this job's output. Say, *id* is a 5-digit *Distributor* job id with leading zeroes, e.g. 00003. Then you will find the job output in `<order dir>/BSI_inp_*/BSI_job_00003`.
- The last field *Done items per job* is only available, if the tool supports the callback mechanism, see 3.5 on page 11. The tool notifies the *Distributor* after each successfully computed item and the number of done items can be reported here. 

If issued from python, an info dictionary with following keys is returned – values are strings if not mentioned below:

- order id
- order dir
- output dir
- input files
- job count
- item count – dictionary, count per job
- bqs job ids
- jobs done
- jobs merged
- jobs removed
- jobs failed
- finished – True or False

6.1.7 Submit <nof jobs>

You can use *Distributor* to just spawn a couple of same jobs via the batch queuing system. The number of jobs to start is passed as the only argument.

This command will create the same directory structure as “normal” orders. A special input file is generated, which contains just the job numbers, each line one number. This file then is splitted using the *AsciiLine splitter*, with *item-size* and *segment-size* set to 1. Each tool will get a segment file as command line argument that contains just one line with the job number.

6.2 Options Reference

We will now first assess the global options which refer to the *Distributor*, and subsequently cover options which relate to the BQS queuing system, used tools and file types. The following sections are set up in an alphabetical order. Beside the respective option names, we have entered the currently valid defaults. So, for example, the default for the option *crontab-empty-msg* is “no crontab for”.

6.2.1 *Distributor* Global Options

config (no default)

Config file for this session, additional to

- /etc/BSI_distributor.cfg,
- \$HOME/.BSI_distributor.cfg and
- ./BSI_distributor.cfg

Cmdline-shortcut: -c

crontab-empty-msg default: no crontab for

If up to now there is no crontab installed for the current user, cron may issue a warning, which is ok to ignore

datetime-format default: %%Y-%%m-%%d-%%H:%%M:%%S

Format for time stamps in files; for format parameters see python's standard module time.

Note: Any percentage sign has to be doubled in config files.

distributor-exec default: BSI_distributor.pyc

Specify main distributor python script. Cron entries then begin with 'python-exec' 'distributor-exec' ; see also python-exec

email default: root

Email recipient for non interactive messages

log default: ./BSI_distributor.log

Specify main log file; you will find additional log info in each order directory

Cmdline-shortcut: -l (ell)

mail-interval default: 10

Interval (in minutes) to get informed about changes via email; 0: don't send emails

max-file-size default: 2147483648 = 2^{31} = 2 GB

Limit of output file size

merger-interval default: 3

Time period (in minutes) between two merger runs

merger-timeout default: 45

After that time period (in minutes) a running merger process is assumed to "hang" and will be deleted

no-merge default: False

If given, no merging is initiated after having submitted jobs

output default: BSI_output

Directory for merged global output files

Cmdline-shortcut: -o

print-dir default: `False`

Print only <order dir> on stdout

Cmdline-shortcut: `-D`

print-id default: `False`

Print only <order id> on stdout. We recommend using `-print-dir!`

Cmdline-shortcut: `-I`

python-exec default: `python`

Python interpreter, may include path

segment-size default: `200`

Number of items per job

Cmdline-shortcut: `-s`

ssh default: `ssh`

Remote shell tool

tmp-dir default: `BSI_tmp`

Directories for order input and local output files. First directory is used like the others but additionally as “master” to store order status info. All directories have to be reachable by NFS from each compute node. Multiple directory entries are separated by Python’s `os.path.pathsep`, which is defined in Unix/Linux as colon ‘:’.

6.2.2 FileType Options

filetype-section default: `mol2`

Type of input files, reference section [FileType <xyz>] in configuration file

Cmdline-shortcut: `-F`

comment-char default: `#`

Character that introduces comment lines in input files

item-size default: `1`

One item consists of ‘item-size’ lines

pattern default: `\\$\\$\\$\\$`

Pattern (regular expression with syntax of Python’s standard module `re`) to split between items.

no-keep-pattern default: `False`

Default is to keep pattern between items in a segment file. Use this option to turn off.

keep-pattern-before default: `False`

Put pattern before first item in segment. Default: don’t put pattern before first item.

keep-pattern-after default: `False`

Put pattern after last item in segment. Default: don’t put pattern after last item.

6.2.3 Tool Options

tool-section default: FlexX

Used tool, reference section [Tool <xyz>] in configuration file

Cmdline-shortcut: -T

segment-placeholder default: @

If found in tool, replaced by segment file name

Cmdline-shortcut: -P

tool default: flexx -b %(flexx-script)s -a '\$(input)=@'

Full commandline inclusive all options

flexx-script default: dock.bat

Batch script for FlexX

6.2.4 BQS Options

bqs-section default: OpenPBS

Used batch queuing system, reference section [BQS <xyz>] in configuration file

Cmdline-shortcut: -B

bqs default: qsub

Full commandline inclusive all options, but without script name to execute

qdel default: qdel

Bqs command to delete a job

qstat default: qstat

Bqs command to get information for all jobs

bqsqueueid (no default)

This option is used indirectly as a parameter to bqs, qdel and qstat; you may want to define bqs as "qsub %(bqsqueueid)s" and specify at command line `-bqsqueueid='-q parallel'`

Cmdline-shortcut: -q