IBM Platform Symphony RFC4501 Readme File

About the "Support for Python v2.7.2"
This package contains the Symphony Standard Python Client API support, which enables the Python application to leverage the Symphony Python API to write Python clients to access C++ services. It also conatins examples that demonstrate the API usage.
The package also includes python wrapper features, which enable Python application code to run task functions on either the local side or the Symphony service side.

Readme file for: IBM® Platform Symphony
Product/Component Release: Symphony 6.1.1
Update Name: Support for Python v2.7.2
Fix ID: sym-6.1.1-build-231678-pimco
Publication date: 8 Apr 2014
Last modified date: 4 May 2014

1 Scope....2
2 Configuration to enable Symphony Python wrapper support....3
2.1 Prerequisites....3
2.2 Installation files....3
2.3 Installation procedure....4
2.4 Configuration procedure....5
2.5 Verification procedure....9
Run basic workload pattern....9
Run parent/child job pattern....10
3 Configuration to enable Symphony Python Client API....10
3.1 Prerequisites....10
3.2 Installation files....10
3.3 Installation procedure....11
3.4 Configuration procedure....11
3.5 Verification procedure....12
Run SampleApp example....12
Run CrossLanguage example....14
Run SharingData example....14
3.6 Review cross language message class....15
4 Usage....15
4.1 How this feature works....15
4.2 Libraries....16
4.3 Environment variables....16
4.4 User interface....16
4.5 Examples....16
Note....16
5 Troubleshooting....17
5.1 Log files....17
6 Copyright and trademark information....18

1 Scope

Applicability	
Operating system	Linux2.6-glibc2.3-x86_64
Python version	2.7.2 64 bit
Symphony version	6.1.1
Cluster types	This feature applies to a single grid cluster or DE.
Other	This feature applies to SOAM.
Dependencies	
File system	This feature has no requirement on the file system type.
<Other>	Python 2.7.2 64 bit version must be installed on the client host and all

	compute hosts.
Limitations	
<Limitation>	a. The Python wrapper is not thread-safe. The user must ensure the methods of a Client object are called in a single thread. b. On the service side, stdout and stdin data files are used to exchange data in a special format. An echo message to stdout will break the protocol.
Known Issues	N/A

2 Configuration to enable Symphony Python wrapper support

2.1 Prerequisites

Python 2.7.2 64-bit version must be installed on the client host and all compute hosts.

2.2 Installation files

This package includes the following files:

File name	Description
`grid/client.py`	Client wrapper to make task functions run on the local side or to submit tasks to make the task functions run on the service side
`grid/dataobjects.py`	Internal module that enables the transfer of task function-related information between the Symphony client and service
`grid/functionrunner.py`	Internal module for running task functions
`grid/__init__.py`	Internal module for the grid package on the client side
`grid/interface/soamapi.pyc`	Internal module for supporting the submission of task functions from the client side and running task functions on the service side
`grid/interface/SoamFactory.so`	Internal library for supporting the submission of task functions from the client side and running task functions on the service side
`grid/cshrc.python; grid/profile.python`	Shell file for setting the PYTHONPATH on the client side

File name	Description
grid/PythonApp.xml	Application profile for this wrapper
grid/PythonApp-1.xml	Application profile for this wrapper for demonstrating the parent/child job pattern
grid/service/__init__.py	Internal module for the grid package on the service side
grid/service/pythonwrapperservice.py grid/service/invokeservice.py	Service wrapper for running task functions on the service side
grid/service/servicerunner.py	Internal module for running task functions on the service side
grid/service/makepackage.sh	Shell file for automatically deploying the service package to the Symphony grid
grid/service/makepackage-1.sh	Shell file for automatically deploying the service package to the Symphony grid for demonstrating the parent/child job pattern
grid/docs/client.html	Client module API reference
grid/samples/test_map_apply.py	Sample module demonstrating how to use the Client.map_apply method
grid/samples/test_imap_apply.py	Sample module demonstrating how to use the Client.imap_apply method
grid/samples/test_apply.py	Sample module demonstrating how to use the Client.apply method
grid/samples/test_parent_child_job.py	Sample module demonstrating how to create parent/child job pattern
grid/samples/test_config_dict_attrs.py	Sample module demonstrating how to use the new config_dict attributes introduced in drop 2

2.3 Installation procedure

1. Download the `sympython-lnx26-lib23-x64-6.1.1.tar.gz` package from the Platform web site.
2. Uncompress the package: `sympython-lnx26-lib23-x64-6.1.1.tar.gz`
   ```
   tar xzvf sympython-lnx26-lib23-x64-6.1.1.tar.gz
   ```

2.4 Configuration procedure

Configuration source	Setting	Behavior
Edit `cshrc.python` and `profile.python`	Replace the "@GRID_DIR_LOCATION@" with the absolute path where the grid dir is located. For example, if the installation package is extracted to /opt/,replace the "@GRID_DIR_LOCATION@" with "/opt".	After sourcing the shell file in the client, PYTHONPATH will point to the directory where the grid directory is located.
`PythonApp.xml`	If you want to change the name of the application, change the `applicationName` attribute in the Consumer section to your desired name. The current `applicationName` is "PythonApp".	After registering this profile, the application with this name will be registered.
	Change the `SessionTypes` settings, including the names of the session types and the detailed configuration of the session types, by configuring the settings in PMC or by editing the application profile manually. To configure the settings in PMC: 1. Log on to the PMC. 2. Go to **Symphony Workload** > **Configure Applications** > **PythonApp** > **Session Type Definition.** 3. Complete your desired configuration in the "Session Type Definition" region. 4. Click the **Save** button in the lower left corner to make your configuration take effect. To configure the settings manually: 1. Open the application profile. 2. Edit the `SessionTypes` section and complete your desired configuration. 3. soamreg the application profile to make your configuration take effect.	**Note**: If the names of session types are customized, remember to adjust your `config_dict['session_type']` setting for the client wrapper on the client side. Otherwise, if the `config_dict['session_type']` does not match any session type name configured in the application profile, the client wrapper cannot submit the tasks to the Symphony grid.

Configuration source	Setting	Behavior
	If Python is not installed in `/usr/local/bin/`, change the value of env "`PATH`" in `Service->osTypes->osType` to where the python is installed on the compute hosts.. The current value of env "`PATH`" is "`/usr/local/bin/`" For example, if Python is installed under `/home/dev/`, change the value of env "`PATH`" to "`/home/dev/`" To configure the setting in PMC: 1. Log on to the PMC. 2. Go to **Symphony Workload** > **Configure Applications** > **PythonApp** > **Service Definition** > **Operating System Definition** > **Environment Variables.** 3. Complete your desired configuration in the "Environment Variables" region. 4. Click the **Save** button in the lower left corner to make your configuration take effect. To configure the settings manually: 1. Open the application profile. 2. Edit the `Service > osTypes > osType > env` section and complete your desired configuration. 3. soamreg the application profile to make your configuration take effect.	After registering this profile, the new env "`PATH`" will take effect. **Note**: The env "`PATH`" must be correctly set to point to the location where Python is installed. Otherwise, the service wrapper will not work.

Configuration source	Setting	Behavior
	If you changed the `Consumer > applicationName`, remember to adjust the `Service > osTypes > osType > fileNamePattern` accordingly. Otherwise, the PMC cannot retrieve the logs generated by the service wrapper. The current fileNamePattern is "`PythonApp_`". For example, if you change `applicationName` to "`MyPythonApp`", adjust the `fileNamePattern` to "`MyPythonApp_`" To configure the setting in PMC: 1. Log on to the PMC. 2. Go to **Symphony Workload** > **Configure Applications** > **PythonApp** > **Service Definition** > **Operating System Definition** > **Logging** > **Log file naming convention (if used)**. 3. Complete your desired configuration in the "Log file naming convention (if used)" text box. 4. Click the **Save** button in the lower left corner to make your configuration take effect. To configure the settings manually: 1. Open the application profile 2. Edit the `Service > osTypes > osType > fileNamePattern` attribute and complete your desired configuration. 3. soamreg the application profile to make your configuration take effect.	After registering this profile, the new fileNamePattern of the Service will take effect. PMC will retrieve the service wrapper log files with the new file name pattern.

Parent-Child Configuration Best Practices

By default, both the parent application and the child application consume resources from the same resource group. Once a parent task is running, it occupies a resource until all of its child tasks are complete. This dependency on the child workload means that the application is blocked from making progress until there is at least one resource available to run child tasks. If the child tasks require resources and its resources are lent out, the resources will be reclaimed with the configured grace period

according to the resource plan. If the reclaim grace period is lengthy, the application might be blocked for a long time. To mitigate potential blockage, it is recommended to apply one of the following best practices:

- Assign some slots to be owned by the child consumer that are not to be lent to other consumers.
- Create separate resource groups for parent workload and child workload so the parent and child never directly compete for resources.

2.5 Verification procedure

Run a basic workload pattern

1. Set up a cluster with only one host and source the `$EGO_TOP/cshrc.platform` in the cluster.
2. Go to the grid directory.
   `cd grid`
3. Edit `cshrc.python` or `profile.python` to replace the "@GRID_DIR_LOCATION@" with the absolute path where the grid directory is located.
4. Source `cshrc.python` or `profile.python`.
5. Create the consumer "`/PythonApp`" in the PMC.
6. Go to the service directory and deploy the service package:
   a) `cd service`
   b) `./makepackage.sh`
7. Register the application:
   `soamreg ../PythonApp.xml`
8. Run the sample test_map_apply.py to test:
   `/usr/local/bin/python ../samples/test_map_apply.py`
   The sample will display the following messages:

```
======================================================================
2011-04-26 20:38:13,726 - PythonWrapperClientLogger - INFO - Connected to
application <PythonApp> connection ID <f9b9ce0a-522f-102e-c000-00123f2ac241-
1105209696-17210>
2011-04-26 20:38:13,741 - PythonWrapperClientLogger - INFO - Created session
with ID <4837>
2011-04-26 20:38:13,765 - PythonWrapperClientLogger - INFO - Done sending <1>
tasks with named_args <{'key': 'value'}>
2011-04-26 20:38:13,766 - PythonWrapperClientLogger - INFO - Fetching task
results ...
Task details:
* task_id < 1 >
* func_name < my_func >
* args < 1 >
* value < 1 >
* task_stats['node'] < devlinux13 >
* task_stats['task_id'] < 1 >
* task_stats['run_time'] < 6.91413879395e-06 >
* exception < None >
* stack_trace < None >
2011-04-26 20:38:15,480 - PythonWrapperClientLogger - INFO - Done fetching
task results ...
2011-04-26 20:38:15,492 - PythonWrapperClientLogger - INFO - Session closed.
2011-04-26 20:38:15,499 - PythonWrapperClientLogger - INFO - Connection
closed.
...
```

Run a parent/child job pattern

This sample demonstrates how to create a parent/child job. It creates two consumers (“/PythonApp “ and “/PythonApp-1”) and registers two applications (“PythonApp” and “PythonApp-1”) under the two consumers separately.

Step 1, 2, 3, 4 are the same as in section 3.5.1.

6. Create the consumer "`/PythonApp`" and "`/PythonApp-1`" in the PMC.
7. Go to the service directory and deploy the service packages for the two consumers:
   ```
   cd service
   ./makepackage.sh
   ./makepackage-1.sh
   ```
8. Register the two applications:
   ```
   soamreg ../PythonApp.xml
   soamreg ../PythonApp-1.xml
   ```
9. Run the sample test_map_apply.py to test:
   ```
   /usr/local/bin/python ../samples/test_parent_child_job.py
   ```
   The sample will display the following messages:

```
======================================================================
2011-07-26 11:14:48,270 - PythonWrapperClientLogger - INFO - Connected to
application <PythonApp> connection ID <57ceb828-9963-102e-c000-00123f2ac241-
1105209696-14068>
2011-07-26 11:14:48,273 - PythonWrapperClientLogger - INFO - Created session
with ID <38507>
2011-07-26 11:14:48,275 - PythonWrapperClientLogger - INFO - Done sending <2>
tasks with named_args <{'key': 'value'}>
2011-07-26 11:14:48,276 - PythonWrapperClientLogger - INFO - Fetching task
results ...
2011-07-26 11:14:55,644 - PythonWrapperClientLogger - INFO - Done fetching
task results ...
2011-07-26 11:14:55,647 - PythonWrapperClientLogger - INFO - Session closed.
2011-07-26 11:14:55,648 - PythonWrapperClientLogger - INFO - Connection
closed.
The final task calculation result value: < 20 >
```

3 Configuration to enable the Symphony Python Client API

3.1 Prerequisites

1. Python 2.7.2 64-bit version must be installed on the client host and all compute hosts.
2. Symphony DE must be installed to run the Cross Language example application. The application requires the C++ sample service in DE.

3.2 Installation files

This package added the following directories based on the Python wrapper package:

File name	Description

`grid/samples/SampleApp`	Sample module demonstrating how to create and run applications with the Symphony standard Python API
`grid/samples/CrossLanguage`	Sample module demonstrating how to create and run applications that the client and service developed with different language
`grid/samples/SharingData`	Sample module demonstrating how to create and run applications that have common data to share

3.3 Installation procedure

1. Download the `sympython-lnx26-lib23-x64-6.1.1.tar.gz` package from the Platform web site.
2. Uncompress the `sympython-lnx26-lib23-x64-6.1.1.tar.gz` package and grid/ directory will be extracted.
   ```
   tar xzvf sympython-lnx26-lib23-x64-6.1.1.tar.gz
   ```

3.4 Configuration procedure

Configuration source	Setting	Behavior
Edit `cshrc.python` and `profile.python`	Replace the "@GRID_DIR_LOCATION@" with the absolute path where the grid directory is located. For example, if the installation package is extracted to /opt/,replace the "@GRID_DIR_LOCATION@" with "/opt".	After sourcing the shell file in the client, PYTHONPATH will point to the directory where the grid directory is located.

3.5 Verification procedure

Run SampleApp example

1. Source the Symphony cluster profile:
   ```
   source $EGO_TOP/cshrc.platform
   ```
2. Go to the grid directory:
   ```
   cd grid
   ```
3. Edit `cshrc.python` or `profile.python` to replace the "@GRID_DIR_LOCATION@" with the absolute path where the grid directory is located.
4. Go to the `SampleApp` example directory:
   ```
   cd samples/SampleApp
   ```
5. Source `cshrc.sampleapp or profile.sampleapp.`

6. Create the consumer "`/SampleAppPython`" in the PMC.
7. Go to the service directory and deploy the service package:
   ```
   cd Service
   ./makepackage.sh
   ```
8. Register the application:
   ```
   soamreg ../SampleAppPython.xml
   ```
9. Run the synchronize sample client:
   ```
   cd ../SyncClient
   python SyncClient.py
   ```

   The sample will display the following messages:

```
========================================================================
[root@vvm SyncClient]# python SyncClient.py
Connected to application: SampleAppPython , Connection ID: 6b71c0bc-0918-
102f-c000-080027ece28d-1118312768-3600
Created session: 5
Sent task: 1
Sent task: 2
Sent task: 3
Sent task: 4
Sent task: 5
Sent task: 6
Sent task: 7
Sent task: 8
Sent task: 9
Sent task: 10
Task Succeeded [ 2 ]
Integer Value :  1
you sent : Hello Grid !!
we replied : Hello Client !!
>>> Synchronously.

Task Succeeded [ 1 ]
Integer Value :  0
you sent : Hello Grid !!
we replied : Hello Client !!
>>> Synchronously.

Task Succeeded [ 3 ]
Integer Value :  2
you sent : Hello Grid !!
we replied : Hello Client !!
>>> Synchronously.

Task Succeeded [ 4 ]
Integer Value :  3
you sent : Hello Grid !!
we replied : Hello Client !!
>>> Synchronously.

Task Succeeded [ 6 ]
Integer Value :  5
you sent : Hello Grid !!
we replied : Hello Client !!
>>> Synchronously.
```

```
Task Succeeded [ 5 ]
Integer Value :  4
you sent : Hello Grid !!
we replied : Hello Client !!
>>> Synchronously.

Task Succeeded [ 7 ]
Integer Value :  6
you sent : Hello Grid !!
we replied : Hello Client !!
>>> Synchronously.

Task Succeeded [ 8 ]
Integer Value :  7
you sent : Hello Grid !!
we replied : Hello Client !!
>>> Synchronously.

Task Succeeded [ 9 ]
Integer Value :  8
you sent : Hello Grid !!
we replied : Hello Client !!
>>> Synchronously.

Task Succeeded [ 10 ]
Integer Value :  9
you sent : Hello Grid !!
we replied : Hello Client !!
>>> Synchronously.

Closing session
Closing connection
All done!!!
```

10. Run the asynchronize sample client:

```
cd ../AsyncClient
python AsyncClient.py
```

Run CrossLanguage example

1. Build the C++ service in DE and deploy to the cluster.
   a) Source the Symphony DE environment:
      ```
      cd /opt/symphonyDE/DE611
      source conf/cshrc.symclient
      ```
   b) Compile the Symphony DE C++ cross language sample:
      ```
      cd 6.1.1/samples/CrossLanguage/CPP
      make
      ```
   c) Create the consumer "/CrossLanguage" in the cluster:
   d) Deploy the C++ service to the cluster:
      ```
      cd Output/
      tar czvf CrossLanguageServiceCPP.tar.gz CrossLanguageServiceCPP
      source $EGO_TOP/cshrc.platform (the cluster environment)
      soamdeploy add CrossLanguageServiceCPP -p
   CrossLanguageServiceCPP.tar.gz -c CrossLanguage
      ```
   e) Register the C++ application service:

```
cd ../..
soamreg CrossLanguageCpp.xml
```

2. Run the Python client to invoke the C++ service.
   a) Go to the grid directory:
      ```
      cd grid
      ```
   b) Edit `cshrc.python` or `profile.python` to replace the "@GRID_DIR_LOCATION@" with the absolute path where the grid dir is located.
   c) Go to the `CrossLanguage` example directory:
      ```
      cd samples/CrossLanguage
      ```
   d) Source `cshrc.crosslanguage or profile.crosslanguage.`
   e) Run the cross language Python client:
      ```
      cd Client
      python CrossLanguageClient.py
      ```
   The Python client will invoke the C++ service.

Run SharingData example

1. Source the Symphony cluster profile:
   ```
   source $EGO_TOP/cshrc.platform
   ```
2. Go to the grid directory:
   ```
   cd grid
   ```
3. Edit `cshrc.python` or `profile.python` to replace the "@GRID_DIR_LOCATION@" with the absolute path where the grid dir is located.
4. Go to the `SharingData` example directory:
   ```
   cd samples/SharingData
   ```
5. Source `cshrc.sharingdata or profile.sharingdata.`
6. Create the consumer "`/SharingDataPython`" in the PMC.
7. Go to the service directory and deploy the service package:
   ```
   cd Service
   ./makepackage.sh
   ```
8. Register the application:
   ```
   soamreg ../SharingData.xml
   ```
9. Run the synchronize sample client:
   ```
   cd ../Client
   python SharingDataClient.py
   ```
10. Review the `MyDataObjects.py` at `grid/samples/SharingData/Common`, `MyCommonData` is the common data to be sharing between tasks.

3.6 Review the cross language message class

1. Review the Python message class at:
   ```
   grid/samples/CrossLanguage/Common/MyMessage.py
   ```
2. Revie the C++ message class at:
   ```
   /opt/symphonyDE/DE611/6.1.1/samples/CrossLanguage/CPP/Common/MyMessage.
h and MyMessage.cpp
   ```
   To pass the message from the Python client to the C++ service, the two message classes must have the same members, methods, and serialize/deserialize implementations.

4 Usage

4.1 How this feature works

This feature provides a client wrapper to run task functions on the local side (local mode) or to submit task functions to the Symphony service side (remote mode). It also provides a service wrapper to run task functions on the Symphony service side.

For local mode, task functions run locally in a sequential way (one by one). For remote mode, task functions run remotely on the Symphony service side in parallel. When task functions are submitted, the task functions and their input are transferred from the client side to the service side. When task function execution finishes on the service side, the calculation output and the related task running information are transferred from the service side back to the client side. On the service side, each service wrapper process will spawn a new child process, which is responsible for the actual task function execution. This occurs when the wrapper process starts for a new session. The wrapper process kills the child process when it finishes serving the session. Spawning the child process will make the PYTHONPATH and LD_LIBRARY_PATH sent from the client side take effect during the task function execution.

The Python wrapper now supports creating parent/child jobs. Client.map_apply() and Client.imap_apply() are extended to accept an optional list of functions (specified as the argument 'dependent_funcs'), which the function pointed to by argument 'func' depends on, either directly or indirectly. The client calls map_apply() or imap_apply() with a parent function as argument 'func' to generate parent tasks. Any functions that the parent function depends on, including the child function, should be passed to map_apply() or imap_apply() as argument 'dependent_funcs'. Any functions that the child function depends on should also be included in this list. When the session is created and tasks are sent, the child function, and the dependent functions of the parent and child functions will be passed from the client side to the service side together with the parent function. This way, the parent function can call its dependent functions during its task function execution and the parent task function can also create child sessions, send child tasks with the child function as argument 'func', and specify the child function's dependent functions as argument 'dependent_funcs' to pass them to the child service instance.

NOTE:
When a client wrapper connects to Symphony, both the user name and password for the connection are "Guest".

4.2 Libraries

`SoamFactory.so` is required by both the client wrapper and the service wrapper.

4.3 Environment variables

The environment variable PYTHONPATH and Symphony-related environment variables must be set correctly to run the python wrapper. Before running the python wrapper, remember to
* source the cshrc.python or profile.python extracted from the installation package.
* source the $EGO_TOP/cshrc.platform in the Symphony installation directory, no matter if local mode or remote mode will be used.

4.4 User interface

Refer to the wrapper API reference client.html located in the grid/docs/ directory.

Refer to the standard Client API reference document:
sym_RFC3385_Standard_Python_Client_API_Reference.doc .

4.5 Examples

1. Uncompress the sympython-lnx26-lib23-x64-6.1.1.tar.gz package:

   ```
   tar xzvf sympython-lnx26-lib23-x64-6.1.1.tar.gz
   ```

2. For further steps, refer to the “Verification procedure” section.

Note

1. The service wrapper must be deployed to the Symphony grid through the `grid/service/makepackage.sh`. Otherwise, the service wrapper might not work as expected.
2. The service wrapper does not handle the service interrupt event that results from suspending the session, killing the session, killing the task, and resource reclaim. In other words, the service wrapper cannot actively react to these events (such as cleaning up resources in the service wrapper, etc.) as soon as the event happens. Set the following *Period parameters in SessionTypes section of the application profile accordingly to make sure these events will not be unexpectedly delayed.
   * For suspending the session: *suspendGracePeriod*
   * For killing the session or killing the task: *taskCleanupPeriod*
   * For resource reclaim: *reclaimGracePeriod*
   For details about these *Period parameters, refer to Knowledge Center > Symphony Reference > Application Profile > SessionTypes section.

5 Troubleshooting

5.1 Log files

Both the client wrapper and the service wrapper have a logging mechanism that is implemented based on the Python logging module.

- There are five logging levels: DEBUG, INFO, WARN, ERROR, and CRITICAL

- The client-side log file name and location depends on the setting to the argument `debug_stdio` of the Client constructor.
  For example, if `debug_stdio` of the Client constructor is set to “`client.log`”, the client.log will be generated under the current working directory.
  For details about how to set the argument `debug_stdio`, refer to the client module API reference in `grid/docs/client.html`.

- Inside a client log file, log messages have the following format:
  ```
  2011-04-17 20:31:15,837 – PythonWrapperClientLogger - DEBUG - ……
  2011-04-17 20:31:15,837 – PythonWrapperClientLogger - INFO - ……
  2011-04-17 20:31:15,837 – PythonWrapperClientLogger - ERROR - ……
  ```

- The service-side log files are generated under the working directory of the service wrapper process. This working directory is determined by the value of the “`workDir`” attribute of `Service->osTypes->osType` section in the application profile. Currently, the “`workDir`” attribute is set to

"$SOAM_HOME/work" in `PythonApp.xml`. For each service wrapper process, there will be two service-side log files generated. Their names have the following format:
* AppName_ServicePID.log: generated by the service wrapper process.
* AppName_ServicePID_WorkerService.log: generated by the child process spawned by the service wrapper process.

For example, a service wrapper process with process id 3456 in PythonApp application will generate the following two log files:
* PythonApp_3456.log
* PythonApp_3456_WorkerService.log

All the service log files (including AppName_ServicePID.log and AppName_ServicePID_WorkerService.log) have file rotation ability. Take PythonApp_3456.log as an example: When it reaches 100 MB, it will be renamed to PythonApp_3456.log.1 and an empty PythonApp_3456.log will be created and start to be written to. When PythonApp_3456.log reaches 100 MB again, the PythonApp_3456.log.1 will be renamed to PythonApp_3456.log.2, the PythonApp_3456.log will be renamed to PythonApp_3456.log.1, and an empty PythonApp_3456.log will be created and start to be written again. The PythonApp_3456.log is always the only file that is being written to.

Currently, for each service log file, the service wrapper can create up to 10 backups. In other words, the maximum suffix number of the log file name is 10. Take PythonApp_3456.log as an example: The backup file name with the maximum suffix is PythonApp_3456.log.10. If PythonApp_3456.log.10 already exists and PythonApp_3456.log reaches 100 MB later, the renaming operation will happen normally as above; the PythonApp_3456.log.10 will be cleared and filled with the previous content of PythonApp_3456.log.9 and no file named "PythonApp_3456.log.11" will be created.

- For each service wrapper process, in addition to the AppName_ServicePID.log and AppName_ServicePID_WorkerService.log, another file called AppName_ServicePID_WorkerService.out is generated (for example, PythonApp_3456_WorkerService.out.). The service wrapper process will redirect anything written to stdout and stderr in the task function to the AppName_ServicePID_WorkerService.out file. This file does not have file rotation ability.

- After the Python wrapper runs for a long time, there might be many service log files generated. To remove these files, go to the directory (by default, $SOAM_HOME/work) specified by the "`workDir`" attribute of `Service > osTypes > osType` section in the application profile and remove as many service log files as you want.

- Inside a service log file, log messages have the following format:

```
2011-04-17 20:31:15,837 - PythonWrapperServiceLogger - DEBUG - ……
2011-04-17 20:31:15,837 - PythonWrapperServiceLogger - INFO - ……
2011-04-17 20:31:15,837 - PythonWrapperServiceLogger - ERROR - ……
```

6 Copyright and trademark information

IBM Web site pages might contain other proprietary notices and copyright information that should be observed.