Broker - Les logs communs des modules du Broker

Sommaire

Libération de ressources lors de la création d'un processus (fork)

Pour exécuter certaines tâches (*Modules*, *Workers*, ...), le Broker doit créer de nouveaux processus sur le système.

Sur les environnements UNIX et notamment Linux, la procédure pour créer un nouveau processus consiste à dupliquer le processus courant. L'appel système associé se nomme **fork**, ainsi, cette action est souvent appelée *un fork*.

Sous Linux, l'opération de **fork** est rapide, l'allocation effective de la mémoire du nouveau processus se faisant lorsque le nouveau processus veut y accéder.

Python disposant d'un *garbage collector*, un outil parcourant la mémoire pour identifier les zones inutilisées et la restituer au système, la mémoire héritée du parent finit par être intégralement copiée dans le nouveau processus.

Pour éviter une consommation inutile de ressources, après le fork, on nettoie les données inutiles issues du processus père (mémoire principalement)

Plus concrètement, le Broker va supprimer les données produites par les modules qu'il a instanciés.

Les modules peuvent fournir une méthode pour gérer la libération de leurs ressources.

Dans tous les cas, le log suivant permet de suivre le nettoyage des ressources :

```
[YYYY-MM-DD HH:MM:SS] INFO: [ MODULE_NAME ] [ CLEAN AFTER FORK ] [ pid:PID] cleanup is starting
[YYYY-MM-DD HH:MM:SS] INFO: [ MODULE_NAME ] [ CLEAN AFTER FORK ] [ pid:PID] cleanup done in X.XXXs
```

Exemple

```
[2021-10-11 06:10:52] INFO   : [ WebUI           ] [ CLEAN AFTER FORK ] [ pid:3736 ] cleanup is starting
[2021-10-11 06:10:52] INFO   : [ WebUI           ] [ CLEAN AFTER FORK ] [ pid:3736 ] cleanup done in 0.143s
```

Une erreur survient pendant le nettoyage des ressources d'un module

Si la méthode de nettoyage d'un module rencontre une erreur, on retrouvera le log qui suit.

Certaines ressources (la mémoire notamment) peuvent ne pas avoir été libérées, mais si le système en a suffisamment de disponibles, Shinken fonctionnera normalement.

```
[YYYY-MM-DD HH:MM:SS] ERROR : [ MODULE_NAME ] [ CLEAN AFTER FORK ] [ pid:PID] [ MODULE_NAME2 ] On linux
system, the forking mechanism (process creation) is fast but create a copy of the father process. So we have
to release unnecessary resources inherited from father. The cleaning has been performed but we encountered
an error:
[YYYY-MM-DD HH:MM:SS] ERROR : [ MODULE_NAME ] [ CLEAN AFTER FORK ] [ pid:PID] [ MODULE_NAME2 ] Cleanup of
data from module  MODULE_NAME2  raised error  IOError occurred
[YYYY-MM-DD HH:MM:SS] ERROR : [ MODULE_NAME ] [ CLEAN AFTER FORK ] [ pid:PID] [ MODULE_NAME2 ] Some memory
may have not been freed. Shinken will still run if enough memory remains available.
[YYYY-MM-DD HH:MM:SS] ERROR : [ MODULE_NAME ] [ CLEAN AFTER FORK ] [ pid:PID] [ MODULE_NAME2 ] You can
report this message to support in order to optimize Shinken memory consumption
[YYYY-MM-DD HH:MM:SS] ERROR : [ MODULE_NAME ] [ CLEAN AFTER FORK ] [ pid:PID] [ MODULE_NAME2 ] ERROR stack :
Traceback (most recent call last):
[YYYY-MM-DD HH:MM:SS] ERROR : [ MODULE_NAME ] [ CLEAN AFTER FORK ] [ pid:PID] [ MODULE_NAME2 ] ERREUR PYTHON
```

Exemple

```
[2021-09-29 15:05:58] ERROR  : [ Livestatus       ] [ CLEAN AFTER FORK ] [ pid:20404 ] [ sla ] On linux
system, the forking mechanism (process creation) is fast but create a copy of the father process. So we have
to release unnecessary resources inherited from father. The cleaning has been performed but we encountered
an error:
[2021-09-29 15:05:58] ERROR  : [ Livestatus       ] [ CLEAN AFTER FORK ] [ pid:20404 ] [ sla ] Cleanup of
data from module  sla  raised error  IOError occurred
[2021-09-29 15:05:58] ERROR  : [ Livestatus       ] [ CLEAN AFTER FORK ] [ pid:20404 ] [ sla ] Some memory
may have not been freed. Shinken will still run if enough memory remains available.
[2021-09-29 15:05:58] ERROR  : [ Livestatus       ] [ CLEAN AFTER FORK ] [ pid:20404 ] [ sla ] You can
report this message to support in order to optimize Shinken memory consumption
[2021-09-29 15:05:58] ERROR  : [ Livestatus       ] [ CLEAN AFTER FORK ] [ pid:20404 ] [ sla ] ERROR stack :
Traceback (most recent call last):
[2021-09-29 15:05:58] ERROR  : [ Livestatus       ] [ CLEAN AFTER FORK ] [ pid:20404 ] [ sla ]   File "/usr
/lib/python2.7/site-packages/shinken/modulesmanager.py", line 877, in do_after_fork_cleanup
[2021-09-29 15:05:58] ERROR  : [ Livestatus       ] [ CLEAN AFTER FORK ] [ pid:20404 ] [ sla ]
inst_cleanup()
[2021-09-29 15:05:58] ERROR  : [ Livestatus       ] [ CLEAN AFTER FORK ] [ pid:20404 ] [ sla ]   File "/var
/lib/shinken/modules/sla/sla_module_broker.py", line 93, in after_fork_cleanup
[2021-09-29 15:05:58] ERROR  : [ Livestatus       ] [ CLEAN AFTER FORK ] [ pid:20404 ] [ sla ]     raise
IOError(u'IOError occurred')
[2021-09-29 15:05:58] ERROR  : [ Livestatus       ] [ CLEAN AFTER FORK ] [ pid:20404 ] [ sla ] IOError:
IOError occurred
```

Envoi et réception de commande entre plusieurs processus

Les logs suivants permettent de suivre l'envoi et la réception de commande se faisant entre plusieurs processus.

Il existe deux types de communications par commande :

- Communication entre le Broker et un de ses modules
- Communication entre un module et un de ses workers

Envoi d'une commande

Warning

Dans le cas d'un premier timeout d'une commande, ce log s'affichera. La commande sera alors renvoyée une deuxième fois.

```
[YYYY-MM-DD HH:MM:SS] WARNING : [ BROKER_NAME ] [ MODULE_NAME ] [ COMMAND CALL ] [ PID:XXXX ] [ THREAD_NAME
] The command call [ COMMAND_NAME ] for module MODULE_NAME was sent, but the call timed out
(TEMPS_TIMEOUTs). Will retry one time.
```

Exemple

```
[2020-11-17 09:12:11] WARNING  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [
CP Server Thread-46 ] The command call [get_module_info] for module Livestatus was sent but the call timed
out (1s). We will retry one time.
```

Il peut arriver qu'à l'envoi d'une commande une autre réponse soit reçue, si la précédente commande n'a pas fonctionné par exemple. Dans ce cas, ce log s'affichera. La commande sera alors renvoyée afin de récupérer la bonne réponse.

```
[YYYY-MM-DD HH:MM:SS] WARNING : [ BROKER_NAME ] [ MODULE_NAME ] [ COMMAND CALL ] [ PID:XXXX ] [ THREAD_NAME ] The command call [ COMMAND_NAME ] was sent but another answer was received. Retrying.
```

Exemple

```
[2020-11-17 09:12:11] WARNING  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ] The command call [get_module_info] was sent but another answer was received. Retrying.
```

Error

Si l'envoi de la commande a connu un premier timeout, à son deuxième elle passera en erreur et ne sera pas renvoyée. Ce log sera affiché :

```
[YYYY-MM-DD HH:MM:SS] ERROR : [ BROKER_NAME ] [ MODULE_NAME ] [ COMMAND CALL ] [ PID:XXXX ] [ THREAD_NAME ] Failed to send command call [ COMMAND_NAME ] for module MODULE_NAME because of timeout (TEMPS_TIMEOUTs).
```

Exemple

```
[2020-11-17 09:12:11] ERROR  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ] Failed to send command call [get_module_info] for module Livestatus because of timeout (1s).
```

Il peut arriver que la commande échoue à cause d'un problème du côté du module/worker. Dans ce cas ce log sera affiché après l'affichage de la stack.

```
[YYYY-MM-DD HH:MM:SS] ERROR : [ BROKER_NAME ] [ MODULE_NAME ] [ COMMAND CALL ] [ PID:XXXX ] [ THREAD_NAME ] Failed to send command call [ COMMAND_NAME ] for module MODULE_NAME because of error: ERROR_MESSAGE
```

Exemple

```
[2020-11-17 09:12:11] ERROR  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ] Failed to send command call [get_module_info] for module Livestatus because of error: 'int' object is not iterable
```

Debug

Dans le cas d'une réussite de l'envoi de la commande, jusqu'à sa réception, ce log s'affichera :

```
[YYYY-MM-DD HH:MM:SS] DEBUG : [ BROKER_NAME ] [ MODULE_NAME ] [ COMMAND CALL ] [ PID:XXXX ] [ THREAD_NAME ] The command call [ COMMAND_NAME ] was executed by the module MODULE_NAME in RUNNING_TIMEs
```

Exemple

```
[2020-11-17 09:12:11] DEBUG  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ] The command call [get_module_info] was executed by the module Livestatus in 0.143s
```

Réception d'une commande

Warning

Si une commande inconnue est reçue, ce log sera affiché :

```
[YYYY-MM-DD HH:MM:SS] WARNING : [ BROKER_NAME ] [ MODULE_NAME ] [ COMMAND CALL ] [ PID:XXXX ] [ THREAD_NAME ] Received unknown command [ COMMAND_NAME ] to execute !
```

Exemple

```
[2020-11-17 09:12:11] WARNING  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ] Received unknown command [get_module_info] to execute !
```

Error

Si la commande crash, ce log sera affiché :

```
[YYYY-MM-DD HH:MM:SS] ERROR : [ BROKER_NAME ] [ MODULE_NAME ] [ COMMAND CALL ] [ PID:XXXX ] [ THREAD_NAME ] Failed to execute received command [ COMMAND_NAME ] with error: ERROR
```

Exemple

```
[2020-11-17 09:12:11] ERROR  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ] Failed to execute received command [get_module_info] with error: Exception
[2020-11-17 09:12:11] ERROR  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ] Traceback (most recent call last):
[2020-11-17 09:12:11] ERROR  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ]   File "C:\dev\workspace\shinken-enterprise\sources\framework\shinken\shinken\basesubprocess.py", line 117, in get_and_execute_command_from_master
[2020-11-17 09:12:11] ERROR  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ]     result = f()
[2020-11-17 09:12:11] ERROR  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ]   File "C:\dev\workspace\shinken-enterprise\testing\test_command_queue_handler.py", line 43, in fail_command
[2020-11-17 09:12:11] ERROR  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ]     raise Exception
[2020-11-17 09:12:11] ERROR  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ] Exception
```

Debug

Au moment de la réception de la commande, si cette dernière est exécutable, ce log sera affiché :

```
[YYYY-MM-DD HH:MM:SS] DEBUG : [ BROKER_NAME ] [ MODULE_NAME ] [ COMMAND CALL ] [ PID:XXXX ] [ THREAD_NAME ] Executing command [ COMMAND_NAME ] with param  PARAMETER_LIST
```

Exemple

```
[2020-11-17 09:12:11] DEBUG  : [ broker-master   ] [ Livestatus       ] [ COMMAND CALL ] [ PID:29341 ] [ CP Server Thread-46 ] Executing command [get_module_info] with param []
```