Documentation de développement de as_run

Table des matières

1 Préambule / Notations

La documentation d'utilisation de `astk` et `as_run` est le document [u1.04.00]. Le présent document s'adresse aux développeurs d'`as_run` et à ceux qui veulent en savoir plus sur la manière dont il fonctionne, et vient compléter les documentations et commentaires présents dans toutes les fonctions du code Python.

Les chemins vers les fichiers/répertoires sont en général relatifs au répertoire d'installation `ASTER_ROOT` (qui vaut `/aster`, `/opt/aster` ou autre).

Lorsqu'on fait référence à un module, on indique son nom en relatif par rapport au package `asrun` ; par exemple `dev/messages.py`.

Pour citer une fonction, on note `common.sysutils.is_localhost` la fonction du module `common/sysutils.py` qui permet de savoir si une machine est la machine courante.

2 Fichiers de configuration

Les fichiers de configuration sont dans `etc/codeaster`. Les informations sont réparties dans :

- **`asrun :`**
  - répertoires de travail,
  - schémas de lancement paramétrables (voir Proxy / Plugins),
  - configuration pour l'interactif, le batch,
  - les lignes de commandes pour le debugger,
  - configuration mpi.
- **`aster :`**
  - définition des versions disponibles,
  - de la version utilisée par défaut.
- **`agla :`**
  - paramètres pour l'AGLA.

Les commentaires présents dans les fichiers précisent le caractère obligatoire/facultatif, et les recommandations à suivre.

Il y a aussi le fichier `config.txt` dans chaque répertoire de version. Voir l'entête du fichier pour la description des paramètres définis. Paramètres particuliers :

- `DEVEL` : permet de dire qu'une version n'est pas surchargeable (C/fortran et catalogue), par exemple, quand il s'agit d'une distribution binaires et/ou sans les bibliothèques. Valeurs : no/yes (yes par défaut)

3 Informations sur le serveur

TODO : que retourne as_run --info

4 Proxy / Plugins

L'option `--proxy` agit, comme son nom l'indique, comme un intermédiaire pour appeler le serveur. Elle est donc appelé du côté client (par l'interface `astk` par exemple). C'est alors `as_run` *client* qui contacte `as_run` *serveur* (qu'il soit sur la même machine ou non).

Prenons l'exemple où un client exécute

```
as_run --proxy --serv filename.export
```

L'option `--proxy` exécute alors la fonction `services.ProxyToServer` qui elle regarde parmi les actions appelables par cet intermédiaire ; actions définies dans le dictionnaire `plugins.actions.ACTIONS`. Pour `--serv`

```
'serv' : {  # --serv .export
    'export_position' : 0,
    'min_args' : 1,
    'max_args' : 1,
    'default_schema' : "asrun.plugins.default.serv",
}
```

On définit le nombre d'arguments et parmi ceux-ci où on trouve le fichier export. Enfin, on définit le *schéma* par défaut, c'est-à-dire la fonction qui doit être utilisée pour exécuter cette action.

Ce fonctionnement permet ainsi de modifier la manière d'exécuter certaines tâches d'où le terme de *plugin* puisqu'il suffit d'ajouter un module exécutant une tâche et d'en indiquer le chemin d'accès (le *schéma*).

Note

Des tâches sont personnalisables à plusieurs niveaux :

- entre le client et le serveur (`--serv`/`schema_serv`, `--actu`/`schema_actu`... : on modifie la manière d'appeler un serveur[1]),
- au niveau de l'appel à un service (`profile_modifier`/`schema_profile_modifier` : on peut agir sur la modification de l'export avant appel au service. Par exemple, passer du service `convbase` à `study`),
- au niveau de l'objet `calcul`/`schema_calcul`, c'est-à-dire au moment de la soumission du service, pour modifier le profil utilisé (voir Exemple de plugin),
- enfin, au moment même de l'exécution, en modifiant l'objet `mpi.MPI_INFO` pour ajuster la ligne de commande (`schema_execute`).

L'appel de la fonction se fait par `services.call_plugin` en utilisant le schéma par défaut ou bien celui fournit en argument par l'option `--schema`. C'est le client qui choisit quel schéma utiliser (l'utilisateur pourrait ainsi choisir d'utiliser telle ou telle méthode).

Le client connait les schémas à utiliser pour chaque action en interrogeant le serveur par `as_run --info` (voir Informations sur le serveur).

> **Note**
> Un plugin peut être dans n'importe répertoire connu du `PYTHONPATH`. Le chemin recommandé est `etc/codeaster/plugins`. Le répertoire `etc/codeaster` est automatiquement mis (par `installation.py`) dans `PYTHONPATH` de manière à définir un schéma du type `plugins.xxx.yyy`, et ce répertoire est conservé lors des mises à jour.

Outre le fait de pouvoir ajouter des plugins, l'utilisation de l'option `--proxy` a permis de repenser les communications client/serveur et de les limiter au sens client > serveur. Une conséquence est, par exemple, que les fichiers résultats d'un calcul ne sont pas automatiquement recopiés mais sur demande (service `--get_results`).

4.1 Description des schémas standards

Le fonctionnement par défaut pour chaque action est défini dans `plugins/default.py`. On présente ici un rapide synopsis de chaque action :

- **`serv`**
  - on copie les fichiers locaux sur le serveur (dans la zone tampon[2]),
  - l'export est adapté en conséquence (l'export original est lui aussi envoyé dans la zone tampon pour usage ultérieur),
  - exécute le service sur le serveur distant.
- **`actu`** (**deux versions sont disponibles en standard :**)
  - `plugins.default.actu` (par défaut, appelle `actu_and_results`) : fait comme `actu_simple` puis demande la recopie des fichiers résultats (`get_results`).
  - `plugins.default.actu_simple` : on lit les arguments et on appelle le service `--actu` sur le serveur.
- **`get_results`**
  - on récupère l'export original,
  - on copie les fichiers résultats sous le nom demandé dans l'export original.
- **`sendmail`**
  - on copie le fichier contenant le mail à envoyer sur le serveur,
  - on appelle le service `--sendmail` sur le serveur.
- **`info`, `del`, `purge_flasheur`, `get_export`**
  - on lit les arguments et on appelle le service correspondant sur le serveur.
- **`edit` (deux versions sont disponibles en standard)**
  - `plugins.default.edit` (par défaut, appelle `local_edit`) : copie le fichier dans le répertoire temporaire local, puis appelle l'éditeur localement (permet de se passer du 'nolisten tcp' de moins en moins répandu).
  - `plugins.default.remote_edit` : ancienne méthode où on ouvre sur le display local l'éditeur distant (ne fonctionne peut-être plus, cf. commentaires).

1. Ce qui n'est sans doute pas très clair c'est que le serveur dit quel *schema* utiliser mais cette *fonction* est exécutée, et donc le code doit être disponible, côté client.

- **tail**
  - on appelle le service `--tail` sur le serveur avec l'option `--result_to_output` (l'argument `fdest` n'est alors pas utilisé).
  - on se contente ensuite d'afficher l'output (qui sera repris par `asjob`).
- **create_issue**
  - on lit le fichier decrivant la fiche et on le copie sur le serveur (dans la zone tampon),
  - si on joint des fichiers, on copie les fichiers en données sur le serveur (dans la zone tampon) ainsi que l'export relocalisé,
  - on appelle le service `--create_issue` sur le serveur.
- **insert_in_db**
  - on copie les fichiers en données sur le serveur (dans la zone tampon) ainsi que l'export relocalisé,
  - on appelle le service `--insert_in_db` sur le serveur.

Le fonctionnement de `serv` où les connexions du serveur vers le client sont nécessaires, est disponible via le schéma `plugins.default.serv_with_reverse_access`.

Pour chaque action, on produit les objets *Server* (cf. Plugins Serveur) à partir de l'export fourni en argument.

> **Note**
> Un paramètre supplémentaire est introduit dans l'export. Il s'agit de `studyid`. C'est une étiquette attribuée sur la machine cliente qui permet d'identifier un calcul, notamment dans la zone tampon. Le `jobid` est l'étiquette attribuée par le serveur lors de la soumission du calcul.

4.2 Plugins Serveur

Aujourd'hui cohabitent deux moyens d'exécuter des commandes ou copier des fichiers sur des machines distantes :
- L'ancienne méthode qui utilise les fonctions `system.AsterSystem.Shell` et `system.AsterSystem.Copy`.
- La nouvelle qui utilise des objets dérivés de `ExecServer`, `CopyToServer`, `CopyFromServer` du module `core/server.py`. Par exemple, `plugins.server.SSHServer` est une déclinaison pour ssh. C'est le serveur (voir Informations sur le serveur) qui dit avec quel type *protocole* il faut le contacter.

A terme, les appels directs à `Shell` et `Copy` du module `system.py` avec des paramètres de machine distantes devraient être résorbés et remplacés par l'utilisation des nouveaux objets.

Ces serveurs utilisent une zone intermédiaire de *transit* pour les fichiers. C'est le paramètre `proxy_dir` du fichier de configuration.

2. Afin d'éviter les conflits de nom pour les fichiers de données et de résultats, on utilise une fonction de conversion (`common.utils.unique_basename`). Cette fonction est utilisée lors du `relocate` du profil dans `plugins.default.copy_datafiles_on_server`, puis lors du rappatriement des fichiers résultats dans le `copyfrom` de `plugins.default.get_result`.

Pour ajouter un nouveau *protocole*, on procède comme pour les actions et on le déclare dans le fichier de configuration `asrun`.

4.3 Exemple de plugin

Sur le serveur de calcul centralisé, on veut personnaliser les paramètres lors de la soumission d'un job. Par exemple, on veut choisir la classe batch selon qu'il s'agit d'une étude standard ou paramétrique, que l'on soit en parallèle ou pas... Pour cela, on a créé un plugin `aster4_calcul.py` qui est placé dans `etc/codeaster/plugins`. Ce module définit une fonction `modifier` qui prend un objet `calcul.AsterCalcul` en argument et retourne un objet `profil.AsterProfil`.

Dans le fichier de configuration, on définit

```
schema_calcul : plugins.aster4_calcul.modifier
```

5 Classes définissant un calcul, un export

TODO A compléter

5.1 Objet `AsterProfil`

Cet objet représente le contenu d'un fichier *export*. Il est constitué de paramètres, d'arguments (de la ligne de commande de Code_Aster) et d'une `collection` (type `EntryCollection`) de fichiers/répertoires (de type `ExportEntry`).

– L'attribut `collection` remplace les attributs `data`/`resu`. Ces derniers sont encore utilisés dans certaines fonctions. Il y a deux fonctions pour assurer la cohérence le temps de la migration. De même, les méthodes `Set`/`Get` vont disparaître.

Le paramètre `tpsjob` est le temps limite du job en minutes. Suite à la conversion, il est utilisé arrondi l'entier supérieur.

Les chemins de fichiers/répertoires (objets `ExportEntry`) peuvent utiliser des variables d'environnement. La fonction `os.path.expandvars` est appelée dans la méthode `get` dans la propriété `path` si le fichier/répertoire est sur la machine locale.

> **Note**
> Le support des variables d'environnement est limité aux objets `ExportEntry`. Le support restera très partiel et localisé à certaines portions du code tant que la migration vers l'objet `EntryCollection` ne sera pas entière.

5.2 Objet(s) AsterCalcul

Il existe en fait plusieurs classes représentant un calcul : AsterCalcul et AsterCalcHandler. Ces deux objets héritent un objet de base BaseCalcul.

– AsterCalcul : Quand on exécute as_run --serv, on crée un objet de ce type qui permet de lancer une exécution standard en soumettant en batch ou interactif un script qui exécute lui-même as_run fichier.export. Si on utilise cet objet dans un script Python (cas de l'astout ou distribution), on peut conserver cet objet pour agir sur le calcul (méthodes get_state, wait, kill, get_diag, etc.).
  À partir de cet classe, on a spécialisé deux classes pour les cas-tests (AsterCalcTestcase) et pour les études paramétriques (AsterCalcParametric) en créant une instance par test ou par variation de l'étude.
– AsterCalcHandler : c'est en quelque sorte, le pendant de AsterCalcul vu du côté du client. On fait donc appel aux fonctions intermédiaires, --proxy, pour agir sur le calcul. Par exemple, là où AsterCalcul.get_state appelle la fonction Func_actu (utilisée par as_run --actu), AsterCalcHandler.get_state passe par le schema_actu (comme as_run --proxy --actu, soit par défaut plugins.default.actu).
  C'est sur la base de cette classe qu'est défini AsterCalcMulti puisque le lancement multiple se place en tant que *client* des différents *serveurs* d'exécution. On crée une instance de AsterCalcMulti par serveur.

> **Note**
> Précaution à prendre quand on crée un objet *calcul* à partir d'un autre : le répertoire de travail étant stocké dans le profil, il faut le supprimer du calcul "esclave" (voir les méthodes change_profile). Sinon le *calcul* esclave utilise le même répertoire et donc le détruit avant que le calcul maître ne soit terminé.

6 Fonctionnement de la distribution de calculs

Le coeur de la distribution est distrib.DistribTask.execute qui surveille l'exécution d'objet dérivé de AsterCalcul.

Il existe deux *timeout* :
– pour débloquer le cas où plus aucun calcul ne peut être soumis (attribut timeout)
– pour avertir qu'un calcul est en machine depuis longtemps (on ne prend pas la décision de le supprimer, on se contente d'avertir).

6.1 Cas particulier des études paramétriques

La classe dérivée utilisée dans ce cas est distrib.DistribParametricTask. On exécute des objets AsterCalcParametric.

6.2 Cas particulier du lancement de cas-tests

La classe dérivée utilisée dans ce cas est `distrib.DistribTestTask`. On exécute des objets `AsterCalcParametric`.

6.3 Cas particulier du lancement multiple

La classe dérivée utilisée dans ce cas est `multiple.DistribMultipleTask`. On exécute des objets `AsterCalcMulti`.

Une exécution *multiple* pouvant être une étude, un lancement de tests..., les résultats produits sont variés : fichiers de message, bases, répertoire de résultats de tests, etc. Les résultats sont donc systématiquement recopiés dans le répertoire `$HOME/MULTI/nomjob_machine` où `nomjob` est le nom de base du profil et `machine` la machine d'exécution. On a donc autant de répertoire de ce type que de machines d'exécution. Dans ces répertoires, on retrouve les fichiers/répertoires résultats (conflit possible si plusieurs résultats ont le même nom de base). De plus, si le paramètre `multiple_result_on_client` est à `yes/oui`, ces répertoires sont *tous* créés sur la machine client. Attention éventuellement au temps de transfert, espace disque.

Exemple d'un profil (syntaxe du fichier export) exécuté depuis `server1` :
– profil initial

```
P actions multiple
P multiple_actions make_etude
P multiple_result_on_client yes
P multiple_server_list server1 server2
P mclient server1
P mode interactif
F comm /opt/aster/NEW11/astest/adlv100a.comm D  1
F mail /opt/aster/NEW11/astest/adlv100a.mail D  20
F resu /tmp/adlv100a.resu R  8
```

– profil sur“server1“

```
P actions make_etude
P mclient server1
P nomjob adlv100a_server1
P serveur server1
P mode interactif
F comm /opt/aster/NEW11/astest/adlv100a.comm D  1
F mail /opt/aster/NEW11/astest/adlv100a.mail D  20
F resu $HOME/MULTI/adlv100a_server1/adlv100a.resu R  8
```

– profil sur `server2`

```
P actions make_etude
P mclient server1
P mode batch
```

```
P nomjob adlv100a_server2
P serveur server2
F comm /tmp/.../bff825ff8667cae36409ed4440d7d8e60e36ca31.adlv100a.comm D
F mail /tmp/.../d3f09aeaca641fa222cfce2181a29657f07d6a6d.adlv100a.mail D
F resu /tmp/.../dc8c57186334bce5283f586eda31e09b4efbc3f7.adlv100a.resu R
```

On voit que l'on passe en batch quand le serveur le permet (ici `server2`). Puisqu'on a demandé les résultats sur la machine cliente (`P multiple_result_on_client yes`), sur `server2`, ils sont produits dans un répertoire intermédiaire. C'est la machine cliente qui viendra chercher les résultats quand l'exécution sur `server2` sera terminée.

Si on avait mis `P multiple_result_on_client no`, on aurait sur `server2`

```
F comm /tmp/.../bff825ff8667cae36409ed4440d7d8e60e36ca31.adlv100a.comm D
F mail /tmp/.../d3f09aeaca641fa222cfce2181a29657f07d6a6d.adlv100a.mail D
F resu $HOME/MULTI/adlv100a_aster4/adlv100a.resu R  8
```

On utilise alors le `$HOME` de `server2`.

7 Gestion de l'exécution des commandes

7.1 Identification des machines

Régulièrement, on a besoin d'exécuter des commandes shell ou de copier des fichiers, et donc de savoir si la machine en question est la machine locale ou un serveur distant. Il y a deux fonctions pour cela :

- `common.sysutils.is_localhost` qui s'appuie uniquement sur le nom de la machine en argument (on peut ignorer ou non le domaine, vérifier que l'on ne change pas de nom d'utilisateur).
- `common_func.is_localhost2` qui appelle la fonction précédente et si le résultat est négatif, fait un test supplémentaire en récupérant le "hostid" de la machine. Elle est dans `common_func` car elle nécessite un objet `run` configuré pour lancer la commande qui est peut-être distante. Le "hostid" n'est pas unique sous Linux (simplement la valeur hexadécimal de la première interface réseau). C'est pour cela qu'on récupère l'adresse MAC en analysant le résultat de `ifconfig` (On pourrait utiliser `uuid.getnode()` si on n'était sûr de tomber sur un python >= 2.5 sur la machine testée).

7.2 Exécution des commandes shell

On utilise `system.AsterSystem` : devrait devenir un simple module.

Aujourd'hui on n'utilise pas `subprocess` car on a besoin de suivre l'output d'un calcul en cours d'exécution. A travailler...

8 Fonctionnalités secondaires

8.1 Log de l'utilisation

Le module `log_usage.py` permet d'enregistrer une trace des exécutions Code_Aster demandées. L'enregistrement est déclenché dans `calcul.AsterCalcul` au moment du lancement effectif (méthode `start`) et à condition que le champ `log_usage_version` soit défini dans le fichier de configuration (`etc/codeaster/asrun` ou `etc/codeaster/agla`).

8.2 Option --quick

Cette option permet de lancer un calcul sans fichier export. Celui-ci est automatiquement généré à partir des noms de fichiers fournis en arguments. On utilise l'extension du nom de fichier pour terminer son type (le numéro d'unité logique associé est celui par défaut).

Les fichiers de données supportés sont les mêmes que ceux cherchés pour lancer un cas-test, soit (dans `execution.py`) : `comm`, `mail`, `mmed`, `med`, `datg`, `mgib`, `msh`, `msup`, `para`, `ensi`, `repe`. À ces types, on peut ajouter les fichiers se terminant par un numéro d'unité logique (exemple : `filename.44`).

Les types de fichiers résultats supportés sont : mess, resu, dat, rmed et les bases. Pour les bases, le nom du répertoire doit se terminer par l'extension `.base` ou `.bhdf`.

L'option `--quick` accepte aussi les options `--surch_pyt` et `--surch_fort` pour exécuter le calcul avec une surcharge.

9 Annexes

9.1 Paramètres du fichier export

– service : nom du service à exécuter

TODO

9.2 Lancement de fonctions sur plusieurs threads

TODO expliquer l'enrobage permettant d'exécuter une fonction en parallèle sur une liste de données et de consolider les résultats "sans risque". Peut-être obsolète (ou à simplifier) avec le module *multiprocessing*.