



Compte Rendu du TP de Coordination et architectures logicielles

Taavi Pehme
Laurent Berthet

Mai 2007

Sommaire

Sommaire	2
Introduction.....	3
1. Architecture des composants	4
1.1 Le lancement et l'activation.....	4
1.2 Le pipe	5
1.3 Le filtre.....	7
1.3.1 L'action du filtre	11
2. Le système complet.....	15
2.1 La communication vers l'extérieur	17
3. Le système de test	18
Conclusion	22
Annexe – le code.....	23

Introduction

L'objectif de ce TP était de réaliser dans un langage parallèle des composants architecturaux relatifs au style « pipe & filter ». Le langage choisi est Erlang pour deux raisons : premièrement, nous l'avons déjà utilisé dans le TP de parallélisme et nous le maîtrisons assez bien ; deuxièmement, le système « pipe & filter » est un système parallèle et Erlang est un langage adapté pour le développement de systèmes parallèles. De plus, nous pouvions utiliser les expériences obtenues dans le TP de parallélisme.

Pour ce TP nous avons créé un projet Eclipse utilisant le plugin de Erlang pour Eclipse. Cela a facilité le développement et c'est facile à utiliser.

Dans ce rapport nous allons présenter l'architecture générale de ces composants (pipe et filter). Nous allons montrer comment les composants sont instanciés et la façon dont ils sont reliés pour créer un système compact. De plus, nous allons présenter un simple système de test créé pour montrer le fonctionnement des composants.

1. Architecture des composants

Dans ce paragraphe nous allons présenter les architectures des composants. On nous a demandé de créer les composants les plus généraux possibles. L'intérêt principal de ca est d'être capable à les utiliser dans plusieurs systèmes. Alors, la pipe et le filtre sont crée les plus généraux possible, la spécification des propriétés est fait par «l'activation» dont on va expliquer dans cette paragraphe.

1.1 Le lancement et l'activation

La pipe et le filtre ont deux états principaux : « *activ* » et « *inactiv* ». Quand un composant est initié comme un processus en utilisant la commande « *spawn* », il est lancé d'abord dans l'état « *inactiv* ». Pour décrire et lancer un processus, le développeur va spécifier seulement quelque paramètres spécifique du composant que nous allons expliquer dans les paragraphes dédiés aux composants spécifiques. Pour le pipe, la fonction de lancement est :

```
loop_pipe(Name, LIST_MAX) -> loop_pipe(Name, null, null, [], LIST_MAX,
inactiv).
```

Avec cette fonction de lancement nous pouvons spécifier pour chaque composant avec qui il peut communiquer. Pour cela, ces « autres composants » que nous passons comme paramètres doivent être déjà créés. Dans un premier temps nous créons tous les composants dans l'état « *inactiv* ». Étant dans cet état le processus ne fait rien d'autre qu'attendre le message d'activation. Une fois tous les processus créé avec « *spawn* », nous allons les activer. Avec le message d'activation nous allons passer aussi une liste de processus – fournisseurs et de processus de sortie. Etant donnée que les processus `PID_PIPE_1_2`, `PID_PIPE_1_3` et `PID_FILTER_1` ont déjà été créé, l'activation de `PID_FILTER_1` sera :

```
PID_FILTER_1!{activate, [PID_PIPE_1_2, PID_PIPE_1_3], [self()]},
```

Où la première liste contient des processus destinataires, et la deuxième contient seulement un processus d'entrée – le système lui-même.

Après avoir été activé, le processus commence à fonctionner.

1.2 Le pipe

Dans notre cas, nous créons les composants pour que les « décisions » sont prises par les filtres, les pipes se comportent de la façon d'une file d'attente. Ils attendent les messages des filtres pour faire des actions. Chaque pipe est connecté à deux filtres. Un de ces deux est l'émetteur, qui fournit les éléments au pipe, le deuxième « consomme » les éléments dans la file d'attente du pipe, fournis par le premier filtre. Pour le pipe, nous spécifions les paramètres suivants :

- **Name** – Le nom du pipe, utilisé pour écrire la trace vers le console. Il est spécifié pendant la création du processus, il ne change pas pendant le fonctionnement.
- **PID_DEST** – L'identificateur du filtre destinataire, spécifié pendant l'activation, il ne change pas pendant le fonctionnement.
- **PID_SRC** – L'identificateur du filtre source, spécifié pendant l'activation, il ne change pas pendant le fonctionnement.
- **LIST_DATA** – La liste des données stockée dans le pipe (la file d'attente) implémentée de la façon FIFO. Il est initié comme une liste vide pendant la création du processus, il change quand les filtres ajoutent ou enlèvent des éléments.
- **LIST_MAX** – Le nombre maximal des éléments stockés dans la liste, il est spécifié pendant la création du processus, il n'est pas changée
- **state** – L'état de la pipe. Peut être soit « activ » ou « inactiv ». Avant l'activation la valeur de state est « inactiv », une fois activé, ca reste « activ ».

Dans l'état « inactiv » le pipe ne peut que s'activer, tous les autres fonctionnalités se passent dans l'état « activ ».

Les messages traités par le pipe sont les suivants :

➤ {PID_SRC_LOC, put, Message}

Dans ce message :

PID_SRC_LOC – L'identificateur du processus qui a envoyé le message
put – Signifie que le message est du type « stocker »
Message – Les données à stocker

Action : On vérifie que l'id d'émetteur est vraiment l'id du **PID_SRC** et que l'état est « activ ». Après on vérifie qu'il y a de la place dans la file d'attente pour stocker l'élément Message. S'il y reste de la place, on stocke Message comme le premier élément de la liste **LIST_DATA** et on notifie l'émetteur que l'émission avait du succès. Si la liste des paramètres est pleine, on notifie l'émetteur que le stockage n'est pas possible.

➤ {PID_DEST_LOC, get}

Dans ce message :

PID_SRC_LOC – L'identificateur du processus qui a envoyé le message
get – Signifie que le message est du type « enlever »

Action : On vérifie que l'id d'émetteur est vraiment l'id du **PID_DEST** et que l'état est « activ ». Après on vérifie que la liste des paramètres n'est pas vide. Si c'est le cas, on va notifier l'émetteur qu'il n'y reste pas des éléments à retirer. Si la liste n'est pas vide, on va envoyer le dernier élément de la liste des paramètres à l'émetteur et on l'enlève de la liste.

➤ {activate, PID_DEST_NEW, PID_SRC_NEW}

Dans ce message :

activate : – Signifie que le message est du type « activation »

PID_DEST_NEW – L'identificateur du processus qui sera le filtre source
PID_SRC_NEW – L'identificateur du processus qui sera le filtre destinataire.

Action : Si l'état du pipe est « inactiv », on va le changer vers « activ » est les variables **PID_DEST = PID_DEST_NEW** et **PID_SRC = PID_SRC_NEW**

➤ {stop}

Dans ce message :

stop : – Signifie que le message est du type « arrêt »

Action : Le processus de la pipe s'arrête.

1.3 Le filtre

Le filtre est la partie du système qui dirige le flux du système. En général les filtres obtiennent des données, ensuite ils exécutent une action sur ces données et finalement ils renvoient les résultats. Un filtre peut avoir plusieurs entrées et plusieurs sorties. Comme les pipes n'ont pas de points communs entre eux, aussi les filtres n'ont pas connectés aux autres filtres, mais qu'à des pipes. Toutes les pipes d'entrée fournissent les paramètres pour l'exécution de l'action spécifique au filtre. Chaque pipe fournit un paramètre. Une fois obtenu, le paramètre est stocké dans la liste des paramètres. Quand tous les paramètres sont obtenus, l'action spécifique au filtre est exécutée et on obtient les résultats sous la forme de liste. Chaque résultat dans cette liste est renvoyé vers un pipe de sortie. Maintenant nous allons expliquer la méthode plus en détail.

Pour le filtre on spécifie les paramètres suivants :

➤ **Name** – Le nom du pipe, utilisé pour écrire la trace vers le console. Il est spécifié pendant la création du processus, il ne change pas pendant le fonctionnement.

- **LIST_PID_DEST** – La liste des identificateurs des pipes destinataires, elle est spécifiée pendant l’activation. Après ca elle ne change pas.
- **LIST_PID_SRC** – La liste des identificateurs des pipes sources, elle est spécifiée pendant l’activation. Après ca elle ne change pas.
- **LIST_PARAMS** – La liste des paramètres. Elle est passée à la fonction spécifique au filtre. Elle est spécifiée comme la liste vide pendant la déclaration du processus, après elle est remplie pendant le fonctionnement du filtre.
- **LIST_RESULTS** – La liste des résultats renvoyée par la fonction. Les éléments de cette liste sont renvoyés aux pipes destinataires.
- **Action** – Spécifie le nom de la fonction spécifique à ce filtre. Toutes les fonctions utilisées doivent être défini dans le module « actions ». Action est spécifiée pendant la déclaration du processus et n’est pas changée après.
- **state** – Spécifie l’état du système. Ca dépend de l’état, quelles actions sont exécutées par le filtre et quels sont les messages traités. On va parcourir la liste des états possibles dans ce paragraphe.
- **Index_in_list_of_PID** – L’index utilisé pour parcourir la liste des pipes sources et la liste des pipes destinataires en les envoyant des messages. Il est initialisé pendant le lancement avec la valeur 0.

Le fonctionnement du filtre est divisé en états différents. Les états sont consécutifs et dans chaque état on fait quelque chose tant que les conditions sont favorables, si les conditions changent, on passe dans un autre état en relançant le boucle avec une nouvelle valeur pour le variable **State**. Ensuite on parcourt la liste des états possible et pour chaque état on spécifie les actions qui peuvent être exécutées dans cet état.

- **Inactiv** – C’est le premier état du filtre après être lancé en utilisant la commande « spawn ». Quand le filtre est dans cet état, il y a le nom, l’action et les deux listes (paramètres et résultats – tous les deux vides) initialisées. Il traite seulement deux messages :
 - {stop} – l’arrêt du processus.
 - {activate, LIST_PID_DEST_NEW, LIST_PID_SRC_NEW}

Dans ce message :

activate - Sa signifie que le message est du type « activation ».

LIST_PID_DEST_NEW – La liste des identificateurs des pipes sources.

LIST_PID_SRC_NEW– La liste des identificateurs des pipes sources destinataire.

Action : Si l'état de la pipe est « inactiv », on va le changer vers « active » est les variables **LIST_PID_DEST = LIST_PID_DEST_NEW** et **LIST_PID_SRC = LIST_PID_SRC_NEW**

➤ **Active** – Après l'activation le filtre passe dans cet état. Il commence son fonctionnement. Dans cette état on parcourt la liste des pipes sources et on les envoie les messages « get » pour obtenir des données. Après avoir envoyé le message à une pipe on passe dans l'état **wait_src** pour attendre le message de cette pipe. Mais si on découvre que tous les pipes d'entrée sont déjà traitées, on passe dans l'état **action**.

➤ **Wait_src** – Dans cet état on attend de la réponse du pipe auquel nous avons envoyé le message. Le pipe peut nous répondre de deux manières possibles :

○ {PIPE_ID, pipe_empty, null}

Dans ce message:

PIPE_ID – L'id du pipe

Pipe_empty – Signifie que le pipe est vide et il n'a rien à envoyer.

Action : On attend un peu (1 seconde), on renvoie la requête au pipe et on reste dans le même état. Donc si on reste bloqué sur un pipe, on va attendre jusqu'à qu'on soit capable d'obtenir les données avant passer à la suivante.

○ {PIPE_ID, pipe_ok, Data}

Dans ce message:

PIPE_ID – L'id de la pipe

Pipe_ok – Signifie que la requête est un succès.

Data – Des données

Action : On stocke les données dans la liste des paramètres sous le même index que l'id de pipe et on passe dans l'état « activ » pour vérifier s'il y reste encore des pipes à consulter.

- **Action** – A ce moment tous les pipes d'entrées sont consultés et les paramètres obtenus, nous allons donc traiter des données. On cherche la fonction avec le nom spécifié dans le variable **Action** dans le module « action ». On passe toute la liste des paramètres et le résultat obtenu, on stocke comme la liste des résultats dans le variable **LIST_RESULTS**. Ensuite on passe dans l'état **send_results**. On va expliquer de la façon détaillée comment créer les fonctions qu'on peut passer comme les actions aux filtres.

- **Send_results** – Quand on passe dans cet état, on a traité des données et obtenu les résultats. Il nous reste à renvoyer les résultats vers les pipes de sorties. On fait la même chose qu'on a fait dans l'état « activ » : on parcourt la liste des pipes destinataires et pour chacun on renvoie l'élément stocké dans la liste des résultats sous le même index que le pipe destinataire dans la liste des pipes. Pour chaque pipe on envoie un message contenant les données et on passe dans l'état **wait_dest** pour attendre la vérification que l'envoi soit réussi. Lors du changement d'état si on rend en compte que tous les résultats sont renvoyés, on passe dans l'état « activ » pour recommencer la procédure avec de nouvelles requêtes vers les pipes sources.

- **Wait_dest** – On attend les vérifications en provenance des pipes destinataires que le renvoi de résultat était réussi. Le pipe peut nous répondre de deux manières possibles :
 - {PIPE_ID, pipe_full}Dans ce message:

PIPE_ID – L'id de la pipe destinataire

Pipe_full – Ca signifie que la pipe est pleine et n'accepte pas des données.

Action : On attend un peu (1 seconde), on renvoie le message à la pipe et on reste dans le même état. Alors, si on reste bloqué sur une pipe, on va attendre jusqu'à elle sera encore capable à recevoir des données avant passer à la pipe suivante.

○ {PIPE_ID, put_ok}

Dans ce message:

PIPE_ID – L'id de la pipe destinataire

put_ok – Ca signifie que le renvoi avait du succès.

Action : On passe dans l'état « send_results » pour vérifier s'il y reste encore des résultats à envoyer.

1.3.1 L'action du filtre

Chaque filtre traite les données qu'il obtient. Il y a une chose qui fait le traitement des données un peu difficile à implémenter : chaque filtre peut prendre un nombre quelconque des paramètres et peut aussi rendre un nombre quelconque des résultats. Pour que le filtre soit le plus général possible, on voulait l'implémenter de la façon qu'en lançant le processus du filtre, on pourrait spécifier la fonction en passant le nom de la fonction comme un paramètre. Il y a des outils dans Erlang pour définir dynamiquement des fonctions, mais la seule chose qui nous a dérangée était le fait que dans ce cas il faut savoir aussi le nombre exact de paramètres et on peut obtenir seulement un résultat. Apparemment, ca ne marcherait pas. Or, nous voulions passer au filtre n'importe quelle fonction avec n'importe quel nombre des paramètres et nous voulions aussi obtenir un nombre quelconque de résultats. Finalement nous avons décidé de faire cette implémentation de la manière le plus générale possible :

Premièrement, on passe seulement le nom de la fonction comme paramètre. Pour cela il faut que toutes les fonctions soient définies dans le module « actions ». Ca nous donne deux avantages : toutes les fonctions sont ensemble et il est plus facile de les manager. Le deuxième avantage est qu'il ne nous est pas nécessaire de spécifier le nom du module comme un paramètre supplémentaire.

Deuxièmement, pour résoudre le problème des paramètres et des résultats, on passe aux toutes les fonctions toute la liste des paramètres. Comme ça, chaque fonctions sait lui-même combien d'éléments de la liste il faut utiliser. Normalement ils doivent tous être utilisés. On a fait la même chose pour les résultats. La fonction d'action va retourner une liste qui est traité comme une liste des résultats par le filtre. Cette façon nous donne une généralité nécessaire. Les éléments de la liste des paramètres et aussi de la liste des résultats peuvent être de n'importe quel type, des constantes, des listes, des listes de listes etc. et leur nombre est illimité.

En utilisant ces deux listes le filtre va agir de la façon suivante : En remplissant la liste des paramètres on sait déjà que chaque élément provient d'un des pipes. Le premier pipe dans la liste des pipes sources va remplir le premier paramètre dans la liste etc. Pour les résultats, on utilise la même logique : Le premier élément dans la liste des résultats est renvoyé vers la première pipe dans la liste de pipes destinataires. Donc c'est au développeur qui code le système en utilisant nos composants de vérifier que les tous les éléments sont bien placés dans les listes et que les fonctions définies pour être utilisées comme les actions utilisent des variables et retournent les résultats de la manière correcte et cohérente.

Exemple :

Admettons qu'on a un filtre qui a un pipe d'entrée et deux pipes de sorties. Le pipe d'entrée doit nous envoyer une liste qui contient des chiffres et des caractères comme le seul paramètre du filtre (on sait que le paramètre peut être de n'importe quel type), puis l'action doit séparer les chiffres et les caractères et nous renvoyer la liste des résultats qui contient deux listes, une pour les chiffres et l'autre pour les caractères.

La liste des paramètres sera : [[1,q,2,w,3,e]]

La fonction qui fait la séparation sur **une liste** sera :

```
separate(List) -> separate(List, [], []).

separate([], List_n, List_l) -> [List_n, List_l];

separate([X|Xs], Ys, Zs) ->
    if erlang:is_number(X) ->
        separate(Xs, [X|Ys], Zs);
    not(erlang:is_number(X)) ->
        separate(Xs, Ys, [X|Zs])
    end.
```

La fonction qui prend la liste de paramètres et qui exécute la fonction de séparation sur le premier élément de la liste sera (On sait qu'on a qu'une seule variable) :

```
separate_a([List|_]) -> separate(List).
```

Quand on lance le filtre, on le passe le nom **separate_a** comme la variable « Action » et dans le filtre la variable « Action » est utilisée comme ça :

```
Fun = {actions, Action},           %% 'actions' est le nom du module,
                                           %% 'Action' = separate_a
```

```
Results = Fun(LIST_PARAMS)    %% 'Results' = [ [3,2,1],[e,w,q] ]  
                               %% LIST_PARAMS = [ [1,q,2,w,3,e] ]
```

2. Le système complet

On a déjà décrit les composants, les modules les plus principaux de notre projet. En fait, pour créer un système utilisant notre composant, on a créé quelques modules supplémentaires qui encapsulent le code et facilitent la conception et le développement.

Les modules supplémentaires sont :

- **Actions** : On a déjà vu que ce module contient des fonctions qui sont appelées dynamiquement par les filtres comme les actions comme **separate_a(List)**. En plus il y a des fonctions qui sont directement utilisées par des fonctions-actions comme **separate(List)**. Comme on a dit avant, toutes les fonctions passées à des filtres comme les actions doivent être définies dans ce module.

- **Util** : Ce module encapsule des fonctions utilitaires classiques qui peuvent servir pour un grand nombre des autres fonctions. On place ici des fonctions de conversion, de traitement des listes. Ici on a créé aussi la fonction « sleep » et une fonction classique qui doit être décrite dans toutes les applications : « log » pour écrire la trace vers « standard output ». Ça nous donne une possibilité à changer les propriétés de « logging » en changeant seulement la fonction « log ».

- **Systeme** : Dans ce module se trouve la composition du système. Les développeurs qui vont créer un système en utilisant nos composants doivent partiellement remplir ce module.
Dans ce module il y a trois méthodes à remplir/changer :
 - **init()** : dans cette méthode on lance, active, et enregistre les composants. Notre logique est que pour le premier filtre qui lance l'enchaînement doit recevoir sa liste de paramètres depuis le système et le dernier filtre doit

passer les résultats au système pour qu'on puisse l'avoir. Alors, pour que ca soit possible il faut pour le premier filtre passer « [self()] » comme la liste des sources, et pour le dernier filtre il faut passer la même chose comme la liste des destinataires.

- **terminate()** : dans cette méthode on traite la situation si le « tour » du système est fait. Ca veut dire que les premiers sets de paramètres et passé et les résultats sont obtenus. Si on ne fait rien dans cette méthode, le système continu à tourner et les filtres continuent à demander les pipes des données. C'est à développeur à décider comment son système doit fonctionner.
 - **start(PARAM)** : en utilisant cette fonction l'utilisateur peut passer les paramètres au premier filtre et obtenir les résultats quand le « tour » est complet. Comme le module « system » est le début et le fin du système, on traite le message de « get » et « put » du premier et du dernier filtre. Les actions qu'on fait après avoir reçu ces messages peuvent être traite volontairement par chaque développeur.
- **Main** : Ce module est utilisé par utilisateur pour lancer le système avec la première liste de paramètres. En principe cette méthode ne doit pas être changée. En ce moment ca fait que appeler « start » et « init » dans le module « system ».

2.1 La communication vers l'extérieur

L'une des questions qui peut se poser est comment peut-on passer les variables au filtre qui va initier l'enchaînement. Pour cela les filtres peuvent communiquer vers l'extérieur. Quand on a réfléchi à cette fonctionnalité, notre but était de maintenir la généralité et la simplicité des filtres. Alors, la solution la plus simple et la plus logique à ce problème était de faire des filtres qui peuvent communiquer vers l'extérieur de la même façon qu'ils communiquent avec les pipes. Donc, quand on définit le système dans le module « system », on rajoute pour les filtres qui doivent communiquer vers l'extérieur, le système lui-même comme un pipe de sortie ou d'entrée. Dans le cas de notre système test on a l'a fait pour le premier et pour le dernier filtre. Le premier reçoit les données depuis le système et le dernier passe les résultats au système. De la côté du système il faut bien sur qu'on traite les messages envoyé par des filtres de la façon qu'on en a besoin.

3. Le système de test

Pour tester les composants et les possibilités de les intégrer dans un système complet on a créé un petit système de test qui fait une tâche classique – le traitement des listes.

Dans ce système on a 5 filtres et 6 pipes qui connectent ces filtres, comme présenté sur le schéma suivant.

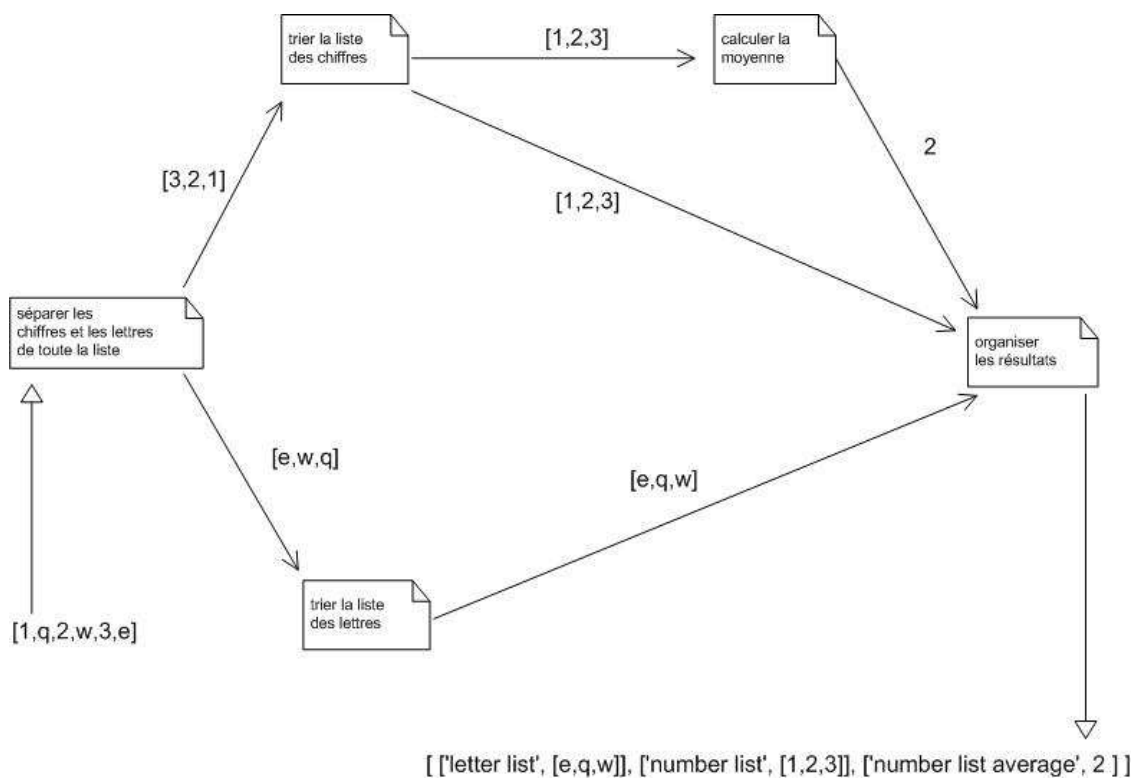


Figure 1 - Le schéma du système test

Ce système reçoit les paramètres depuis l'extérieur et il renvoie les résultats aussi vers l'extérieur. Pour construire le système comme ça on le passe dans la phase d'activation « l'extérieur » comme le pipe d'entrée pour le premier filtre et comme le pipe de sortie pour le dernier filtre. Alors, le premier filtre va recevoir comme toute la liste des

paramètres une liste, qui contient des chiffres et des caractères. Il va séparer les chiffres et les caractères et va créer deux listes comme le résultat, une liste pour chaque type. Puis il va renvoyer les deux listes créées vers les filtres différents.

Le deuxième filtre reçoit la liste des chiffres comme seul paramètre, il va les trier, et il renvoie les résultats vers deux filtres : le filtre final et le calculateur de la moyenne.

Le calculateur de la moyenne reçoit la liste, puis il va calculer la moyenne des éléments de la liste et renvoie le résultat vers la filtre final. Celui qui reçoit les caractères va le trier et les renvoyer vers le filtre final. Quand le filtre final reçoit tous les trois paramètres, il va les organiser dans une liste de la façon spécifiée et il le renvoie vers l'extérieur.

La trace sortie par le système pour le même exemple est la suivante.

```
(Tp_archi@vontaavi)1> main:main([1,q,2,w,3,e]).
f1: 'received activate'
f2: 'received activate'
f3: 'received activate'
f4: 'received activate'
f5: 'received activate'
p1_2: 'received activate'
p1_3: 'received activate'
p2_4: 'received activate'
p3_4: 'received activate'
p2_5: 'received activate'
p5_4: 'received activate'
f1: 'params request sent to', 'Pid:' : <0.39.0>
f2: 'params request sent to', 'Pid:' : <0.56.0>
f3: 'params request sent to', 'Pid:' : <0.57.0>
f4: 'params request sent to', 'Pid:' : <0.59.0>
f5: 'params request sent to', 'Pid:' : <0.60.0>
p1_2: 'received get, pipe_empty'
p1_3: 'received get, pipe_empty'
p3_4: 'received get, pipe_empty'
p2_5: 'received get, pipe_empty'
f1: 'waiting for source, pipe_ok'
f2: 'waiting for source', <0.56.0> : pipe_empty
```

```

f3: 'waiting for source', <0.57.0> : pipe_empty
f4: 'waiting for source', <0.59.0> : pipe_empty
f5: 'waiting for source', <0.60.0> : pipe_empty
f1: 'all params collected', 'Param list' : [[1,q,2,w,3,e]]
f1: 'calculating...', param_list : [[1,q,2,w,3,e]]
f1: 'calculating complete', 'result_list: ' : [[3,2,1],[e,w,q]]
f1: 'sending pram to destination', param : [3,2,1]
p1_2: 'received put, put_ok'
f1: 'destination received data'
f1: 'sending pram to destination', param : [e,w,q]
p1_3: 'received put, put_ok'
f1: 'destination received data'
f1: 'all destinations done'
f1: 'params request sent to', 'Pid:' : <0.39.0>
p2_5: 'received get, pipe_empty'
p3_4: 'received get, pipe_empty'
p1_3: 'received get, data sent'
p1_2: 'received get, data sent'
f5: 'waiting for source', <0.60.0> : pipe_empty
f4: 'waiting for source', <0.59.0> : pipe_empty
f3: 'waiting for source, pipe_ok'
f2: 'waiting for source, pipe_ok'
f3: 'all params collected', 'Param list' : [[e,w,q]]
f2: 'all params collected', 'Param list' : [[3,2,1]]
f3: 'calculating...', param_list : [[e,w,q]]
f2: 'calculating...', param_list : [[3,2,1]]
f3: 'calculating complete', 'result_list: ' : [[e,q,w]]
f2: 'calculating complete', 'result_list: ' : [[1,2,3],[3,2,1]]
f3: 'sending pram to destination', param : [e,q,w]
f2: 'sending pram to destination', param : [1,2,3]
p3_4: 'received put, put_ok'
p2_4: 'received put, put_ok'
f3: 'destination received data'
f2: 'destination received data'
f3: 'all destinations done'
f2: 'sending pram to destination', param : [3,2,1]
f3: 'params request sent to', 'Pid:' : <0.57.0>
p2_5: 'received put, put_ok'

```

p1_3: 'received get, pipe_empty'
f2: 'destination received data'
f3: 'waiting for source', <0.57.0> : pipe_empty
f2: 'all destinations done'
f2: 'params request sent to', 'Pid:' : <0.56.0>
p1_2: 'received get, pipe_empty'
f2: 'waiting for source', <0.56.0> : pipe_empty
p1_2: 'received get, pipe_empty'
p1_3: 'received get, pipe_empty'
p3_4: 'received get, data sent'
p2_5: 'received get, data sent'
f2: 'waiting for source', <0.56.0> : pipe_empty
f3: 'waiting for source', <0.57.0> : pipe_empty
f4: 'waiting for source, pipe_ok'
f5: 'waiting for source, pipe_ok'
f4: 'params request sent to', 'Pid:' : <0.58.0>
f5: 'all params collected', 'Param list' : [[3,2,1]]
p2_4: 'received get, data sent'
f5: 'calculating...', param_list : [[3,2,1]]
f4: 'waiting for source, pipe_ok'
f5: 'calculating complete', 'result_list: ' : [2]
f4: 'params request sent to', 'Pid:' : <0.61.0>
f5: 'sending pram to destination', param : 2
p5_4: 'received get, pipe_empty'
p5_4: 'received put, put_ok'
f4: 'waiting for source', <0.61.0> : pipe_empty
f5: 'destination received data'
f5: 'all destinations done'
f5: 'params request sent to', 'Pid:' : <0.60.0>
p2_5: 'received get, pipe_empty'
f5: 'waiting for source', <0.60.0> : pipe_empty
p2_5: 'received get, pipe_empty'
p5_4: 'received get, data sent'
p1_3: 'received get, pipe_empty'
p1_2: 'received get, pipe_empty'
f5: 'waiting for source', <0.60.0> : pipe_empty
f4: 'waiting for source, pipe_ok'
f3: 'waiting for source', <0.57.0> : pipe_empty

```

f2: 'waiting for source', <0.56.0> : pipe_empty
f4: 'all params collected', 'Param list' : [[e,q,w],[1,2,3],2]
f4: 'calculating...', param_list : [[e,q,w],[1,2,3],2]
f4: 'calculating complete', 'result_list: ' : [[['letter
list',[e,q,w]],['number
list',[1,2,3]],['number list average',2]]]
f4: 'sending pram to destination', param : [['letter
list',[e,q,w]],['number lis
t',[1,2,3]],['number list average',2]]
'system:': 'Results received: '
f4: 'destination received data'
system: [['letter list',[e,q,w]],['number list',[1,2,3]],['number list
average',2]]
f4: 'all destinations done'
f4: 'params request sent to', 'Pid:' : <0.59.0>
{ok}

```

Conclusion

Pendant ce TP on a créé des composants d'architecture du type pipe-&-filter. On a suivi la demande de généralité, donc les composants sont utilisables de différentes manières, par contre, les développeurs qui utilisent ces composants doivent être attentifs car ils ont beaucoup de liberté et les propriétés du système ne sont pas très strictement définies. Pour la technique de développement on a utilisé des choses apprises pendant le TP de parallélisme. Pour tester et présenter le système on a créé un système de traitement de listes qui a bien passé des tests.

Annexe – le code

```
%%% -----
%%% @version 0.1, {@date} {@time}.
%%% @author T. Pehme, L. Berthet
%%% @doc Le module {@module} presente une tube (pipe) qui gere la communication entre
deux filters
%%% @end
%%% -----
-module(pipe).
-export([loop_pipe/2]).

%%-----
%% Le methode qui initialise le tube dans l'etat inactive
%%-----
loop_pipe(Name, LIST_MAX) -> loop_pipe(Name, null, null, [], LIST_MAX, inactiv).

%%-----
%% Le boucle principal
%%-----
loop_pipe(Name, PID_DEST, PID_SRC, LIST_DATA, LIST_MAX, State) ->

    receive
        %% Le message de stockage, si il y reste de la place dans le file
d'attente
        {PID_SRC_LOC, put, Message} when length(LIST_DATA) < LIST_MAX ,PID_SRC_LOC
== PID_SRC ,State == activ ->
            util:log(Name, 'received put, put_ok'),
            PID_SRC!{self(), put_ok},
            New_list_1 = util:add(Message, LIST_DATA),
            loop_pipe(Name, PID_DEST, PID_SRC, New_list_1, LIST_MAX, State);

        %% Le message de stockage, si le file d'attente est plein
        {PID_SRC_LOC, put, _} when length(LIST_DATA) == LIST_MAX ,PID_SRC_LOC == PID_SRC
,State == activ ->
            util:log(Name, 'received put, pipe_full'),
            PID_SRC!{self(), pipe_full},
            loop_pipe(Name, PID_DEST, PID_SRC, LIST_DATA, LIST_MAX, State);

        %% Le message de requete, si le tube est vide
        {PID_DEST_LOC, get} when length(LIST_DATA) == 0 ,PID_DEST_LOC == PID_DEST ,State
== activ ->
            util:log(Name, 'received get, pipe_empty'),
            PID_DEST!{self(), pipe_empty, null},
            loop_pipe(Name, PID_DEST, PID_SRC, LIST_DATA, LIST_MAX, State);

        %% Le message de requete, si le tube n est pas vide
        {PID_DEST_LOC, get} when length(LIST_DATA) > 0 ,PID_DEST_LOC == PID_DEST ,State ==
activ ->
            util:log(Name, 'received get, data sent'),
            Last_element_in_list = lists:last(LIST_DATA),
            New_list_2 = util:remove_last(LIST_DATA),
            PID_DEST!{self(), pipe_ok, Last_element_in_list},
            loop_pipe(Name, PID_DEST, PID_SRC, New_list_2, LIST_MAX, State);

        %% Le message d activation
        {activate, PID_DEST_NEW, PID_SRC_NEW} when State == inactiv ->
            util:log(Name, 'received activate'),
            loop_pipe(Name, PID_DEST_NEW, PID_SRC_NEW, LIST_DATA, LIST_MAX, activ);

        %% Le message pour arreter le processus
        {stop} ->
            exit(normal)
    end.
```

```

%%% -----
%%% @version 0.1, {@date} {@time}.
%%% @author T. Pehme, L. Berthet
%%% @doc Le module {@module} presente une filter qui recoit des donnees, les traite et
les renvoie
%%% @doc vers la tube (pipe) de sortie
%%% @end
%%% -----
-module(filter).
-export([loop_filter/2]).

%%-----
%% Le methode qui initialise le filtre dans l'etat inactive
%%-----
loop_filter(Name, Action) -> loop_filter(Name, null, null, [], [], Action, inactiv, 0).

%%-----
%% Le boucle principal
%%-----
loop_filter(Name, LIST_PID_DEST, LIST_PID_SRC, LIST_PARAMS, LIST_RESULTS, Action, State,
Index_in_list_of_PID) ->
    %% si l'etat est inactive on attend l'activation
    if State == inactiv ->

        receive

            {activate, LIST_PID_DEST_NEW, LIST_PID_SRC_NEW} when
State == inactiv ->
                util:log(Name, 'received activate'),
                loop_filter(Name, LIST_PID_DEST_NEW, LIST_PID_SRC_NEW,
LIST_PARAMS, LIST_RESULTS, Action, activ, Index_in_list_of_PID);
                {stop} ->
                    exit(normal)
        end;

        %% si l'etat est active, la filtre peut commencer a remplir la liste des
parametres
        State == activ ->

            %% si la liste des parametres est plein on va passer dans l'etape de
calcul
            if Index_in_list_of_PID >= length(LIST_PID_SRC) ->
                util:log(Name, 'all params collected', 'Param list',
LIST_PARAMS),
                loop_filter(Name, LIST_PID_DEST, LIST_PID_SRC,
LIST_PARAMS, LIST_RESULTS, Action, action, 0);

                %% si la liste des parametres est pas rempli, on envoie la message au
pipe suivant,
                %% incremente l'indice dans la liste des pipes et pass dans l'etat
d'attente de reponse du pipe
                Index_in_list_of_PID < length(LIST_PID_SRC) ->

                    New_index = Index_in_list_of_PID + 1,
                    PID_IN_LIST = lists:nth(New_index,
LIST_PID_SRC),
                    PID_IN_LIST!{self(), get},
                    util:log(Name, 'params request sent to', 'Pid:',
PID_IN_LIST),
                    loop_filter(Name, LIST_PID_DEST, LIST_PID_SRC, LIST_PARAMS,
LIST_RESULTS, Action, wait_src, New_index)
            end; %%end if

            %% on attend la reponse du pipe
            State == wait_src ->

                %%util:log('wait src'),
                receive

```

```

%% si le pipe peut pas nous fournir les parametres on continue a
attendre
        {PIPE_ID, pipe_empty, null} ->
        util:log(Name, 'waiting for source', PIPE_ID, 'pipe_empty'),
        util:sleep(1000),
        PIPE_ID!{self(), get},
        loop_filter(Name, LIST_PID_DEST, LIST_PID_SRC, LIST_PARAMS,
LIST_RESULTS, Action, State, Index_in_list_of_PID);

%% si les parametres sont fournis, on les stocke et passa au pipe
suivant
        {PIPE_ID, pipe_ok, Data} ->
        util:log(Name, 'waiting for source, pipe_ok'),
        New_list = util:add_last(Data, LIST_PARAMS),
        loop_filter(Name, LIST_PID_DEST, LIST_PID_SRC, New_list,
LIST_RESULTS, Action, activ, Index_in_list_of_PID);

        {stop} ->
                exit(normal)
end;

%% on calcul le resultat et commence a distribuer les elements du resultat aux
pipes
State == action ->
        util:log(Name, 'calculating...', 'param_list', LIST_PARAMS),
        Fun = {actions, Action},
        Results = Fun(LIST_PARAMS),
        util:log(Name, 'calculating complete', 'result_list: ', Results),
        loop_filter(Name, LIST_PID_DEST, LIST_PID_SRC, [], Results, Action,
send_results, 0);

State == send_results ->

d'attente
%% si tous les tubes destinataires sont parcourus, on passe dans l'etat
if Index_in_list_of_PID >= length(LIST_PID_DEST) ->
        util:log(Name, 'all destinations done'),
        loop_filter(Name, LIST_PID_DEST, LIST_PID_SRC, [], [],
Action, activ, 0);

Index_in_list_of_PID < length(LIST_PID_DEST) ->
        New_index = Index_in_list_of_PID + 1,
        PID_IN_LIST = lists:nth(New_index,
LIST_PID_DEST),
        RESULT_IN_LIST = lists:nth(New_index, LIST_RESULTS),
        PID_IN_LIST!{self(), put, RESULT_IN_LIST},
        util:log(Name, 'sending pram to destination', 'param',
RESULT_IN_LIST),
        loop_filter(Name, LIST_PID_DEST, LIST_PID_SRC, LIST_PARAMS,
LIST_RESULTS, Action, wait_dest, New_index)

end; %%end if
State == wait_dest ->

receive
%% si le pipe destinataire peut pas accepter le message on
continue a attendre
        {PIPE_ID, pipe_full} ->
        util:log(Name, 'destination pipe full'),
        util:sleep(1000),
        RESULT_IN_LIST = lists:nth(Index_in_list_of_PID,
LIST_RESULTS),
        PIPE_ID!{self(), put, RESULT_IN_LIST},
        loop_filter(Name, LIST_PID_DEST, LIST_PID_SRC, LIST_PARAMS,
LIST_RESULTS, Action, State, Index_in_list_of_PID);

%% si les resultats sont envoyees, on passe au pipe suivant

```

```

        {PIPE_ID, put_ok} ->
            util:log(Name, 'destination received data'),
            loop_filter(Name, LIST_PID_DEST, LIST_PID_SRC, LIST_PARAMS,
LIST_RESULTS, Action, send_results, Index_in_list_of_PID);

        {stop} ->
            exit(normal)

    end

end. %%end if

%%% -----
%%% @version 0.1, {@date} {@time}.
%%% @author T. Pehme, L. Berthet
%%% @doc Le module {@module} presente le systeme qui encpsule/contient des pipes et des
filtres, et qui
%%% @doc communique avec l'exterieur via "input" et "output"
%%% @end
%%% -----
-module(system).
-export([init/0, start/1, terminate/0]).

%%-----
%% Le methode qui declare, active et enregistre les composant
%%-----
init() ->

    %% declaration
    PID_FILTER_1 = spawn_link(filter, loop_filter,[f1, separate_a]),
    PID_FILTER_2 = spawn_link(filter, loop_filter,[f2, sort_and_original_a]),
    PID_FILTER_3 = spawn_link(filter, loop_filter,[f3, sort_a]),
    PID_FILTER_4 = spawn_link(filter, loop_filter,[f4, complete_a]),
    PID_FILTER_5 = spawn_link(filter, loop_filter,[f5, average_a]),

    PID_PIPE_1_2 = spawn_link(pipe, loop_pipe,[p1_2, 10]),
    PID_PIPE_1_3 = spawn_link(pipe, loop_pipe,[p1_3, 10]),
    PID_PIPE_2_4 = spawn_link(pipe, loop_pipe,[p2_4, 10]),
    PID_PIPE_3_4 = spawn_link(pipe, loop_pipe,[p3_4, 10]),
    PID_PIPE_2_5 = spawn_link(pipe, loop_pipe,[p2_5, 10]),
    PID_PIPE_5_4 = spawn_link(pipe, loop_pipe,[p5_4, 10]),

    %% activation
    PID_FILTER_1!{activate, [PID_PIPE_1_2, PID_PIPE_1_3], [self()]},
    PID_FILTER_2!{activate, [PID_PIPE_2_4, PID_PIPE_2_5], [PID_PIPE_1_2]},
    PID_FILTER_3!{activate, [PID_PIPE_3_4], [PID_PIPE_1_3]},
    PID_FILTER_4!{activate, [self()], [PID_PIPE_3_4, PID_PIPE_2_4, PID_PIPE_5_4]},
    PID_FILTER_5!{activate, [PID_PIPE_5_4], [PID_PIPE_2_5]},

    PID_PIPE_1_2!{activate, PID_FILTER_2, PID_FILTER_1},
    PID_PIPE_1_3!{activate, PID_FILTER_3, PID_FILTER_1},
    PID_PIPE_2_4!{activate, PID_FILTER_4, PID_FILTER_2},
    PID_PIPE_3_4!{activate, PID_FILTER_4, PID_FILTER_3},
    PID_PIPE_2_5!{activate, PID_FILTER_5, PID_FILTER_2},
    PID_PIPE_5_4!{activate, PID_FILTER_4, PID_FILTER_5},

    %% enregistrement
    register(filter_1,PID_FILTER_1 ),
    register(filter_2,PID_FILTER_2 ),
    register(filter_3,PID_FILTER_3 ),
    register(filter_4,PID_FILTER_4 ),
    register(filter_5,PID_FILTER_5 ),
    register(pipe_1_2,PID_PIPE_1_2),
    register(pipe_1_3,PID_PIPE_1_3),

```

```

    register(pipe_2_4,PID_PIPE_2_4),
    register(pipe_3_4,PID_PIPE_3_4),
    register(pipe_2_5,PID_PIPE_2_5),
    register(pipe_5_4,PID_PIPE_5_4).

%%-----
%% Le methode qui termine les procesuses
%%-----
terminate() ->
    filter_1!{stop},
    filter_2!{stop},
    filter_3!{stop},
    filter_4!{stop},
    pipe_1_2!{stop},
    pipe_1_3!{stop},
    pipe_2_4!{stop},
    pipe_3_4!{stop},
    filter_5!{stop},
    pipe_2_5!{stop},
    pipe_5_4!{stop}.

%%-----
%% Le methode qui lance le systeme avec les parametres d'utilisateur
%%-----
start(PARAM_FOR_FIRST_FILTER) -> start(PARAM_FOR_FIRST_FILTER, state1).

start(PARAM_FOR_FIRST_FILTER, State) ->
    receive
        {PID_FILTER_1, get} when State == state1 ->
        PID_FILTER_1!{self(), pipe_ok, PARAM_FOR_FIRST_FILTER},
        start(PARAM_FOR_FIRST_FILTER, state2);

        {PID_FILTER_4, put, RESULT_IN_LIST} when State == state2 ->
        PID_FILTER_4!{self(), put_ok},
        util:log('system:', 'Results received: '),
        util:log('system', RESULT_IN_LIST),
        terminate(),
        {ok}

    end.

%%% -----
%%% @version 0.1, {@date} {@time}.
%%% @author T. Pehme, L. Berthet
%%% @doc Le module {@module} contient des fonctions utilitaires
%%% @end
%%% -----
-module(util).
-export([add/2, add_last/2, remove_last/1, log/2, log/4, sleep/1]).

%%-----
%% Args: E - element, L - liste
%% Rajout l'element E dans la tete de la liste L
%%-----

add(E,L) -> [E|L].

%%-----
%% Args: E - element, L - liste
%% Rajout l'element E comme le dernier element de la liste L
%%-----

add_last(E,L) ->
    lists:append(L, [E]).

```

```

%%-----
%% Args: L - liste
%% Supression d'element dans la tete de la liste L
%% Return: La nouvelle liste
%%-----

%% dans le ca d'une liste vide on fait rien
remove_last([]) ->
  [];

%%dans le ca d'une liste pas vide on supprime le dernier element
remove_last([X|Xs]) ->
  lists:reverse(tl(lists:reverse([X|Xs]))).

%%-----
%% Args: Message - le message a logger
%% On va logger le message dans le sortie par default
%%-----

log(Object, Message) ->
  io:format("~w: ~w \n" , [Object, Message]).

log(Object, Message, Message2, Message3) ->
  io:format("~w: ~w, ~w : ~w \n" , [Object, Message, Message2, Message3]).

%%-----
%% Args: Millis - le nombre de millisecondes a attendre
%% La fonction "sleep"
%%-----

sleep(Millis) ->
  receive
    after Millis -> ok
  end.

%%% -----
%%% @version 0.1, {@date} {@time}.
%%% @author T. Pehme, L. Berthet
%%% @doc Le module {@module} contient des actions executees par les filtres
%%% @end
%%% -----
-module(actions).
-export([sort/1, delete_numbers_a/1, sort_a/1, delete_numbers/1, add_begin_end/1,
separate_a/1, flatten_a/1, first_a/1, last_a/1, average_a/1, complete_a/1,
sort_and_original_a/1]).

%%-----
%% On va trier la liste de la facon normal
%% Args: List - La liste des elements
%% Return : La liste contenant les elements tries de la liste initiale
%%-----

sort_a([X|_]) ->
  [sort(X)].

sort(List) -> lists:sort(List).

%%-----

```

```

%% On va trier la liste de la facon normal et on renvoie la liste triée et la liste
originale
%% Args: List - La liste des elements
%% Return : La liste contenant les elements tries ET la liste initiale
%%-----

sort_and_original_a([X|_]) ->
    [sort(X), X].

%%-----
%% On va enlever les nombres de la liste
%% Args: List - La liste des elements
%% Return : La liste contenant les elements qui sont pas les nombres de la liste initiale
%%-----

delete_numbers_a([X|_]) ->
    [delete_numbers(X)].

delete_numbers([]) -> [];

delete_numbers([X|Xs]) ->
    if erlang:is_number(X) ->
        delete_numbers(Xs);
    not(erlang:is_number(X)) ->
        [X|delete_numbers(Xs)]
    end.

%%-----
%% On va separer les nombres et les lettres dans la liste
%% Args: List - La liste des elements
%% Return : La liste contenant deux listes: un pour les nombres et l autre pour les
lettres
%%-----

separate_a([List|_]) -> separate(List).

separate(List) -> separate(List, [], []).

separate([], List_n, List_l) -> [List_n, List_l];

separate([X|Xs], Ys, Zs) ->
    if erlang:is_number(X) ->
        separate(Xs, [X|Ys], Zs);
    not(erlang:is_number(X)) ->
        separate(Xs, Ys, [X|Zs])
    end.

%%-----
%% On va rajouter les identifiateur BEGIN et END au debut et a al fin de la liste
%% Args: List - La liste a modifier
%% Return : La liste: [BEGIN, ... <elements> ... , END]
%%-----

add_begin_end(List) ->
    lists:append(['BEGIN'|List], ['END']).

%%-----
%% On utilise flatten sur toute la liste de parametres
%% Args: List - La liste des parametres
%% Return : La liste normal: [...]
%%-----

flatten_a(List) ->
    [lists:flatten(List)].

```

```

%%-----
%% On prend le premier element de la liste qui est le premier element de la liste des
parametres
%% Args: List - La liste des parametres
%% Return : Element
%%-----

first_a([X|_]) ->
    [hd(X)].

%%-----
%% On prend le dernier element de la liste qui est le premier element de la liste des
parametres
%% Args: List - La liste des parametres
%% Return : Element
%%-----

last_a([X|_]) ->
    [lists:last(X)].

%%-----
%% On calcule la moyenne de tous les elements de la liste qui est le premier argument de
la liste des parametres
%% Args: List - La liste des parametres
%% Return :La moyenne avec la précision d'entier
%%-----
average_a([List|_]) -> [average(List)].

average(List) -> erlang:round(lists:sum(List) / length(List)).

%%-----
%% On rearranger les trois paramètres dans la liste et on va les sortir
%% Args: List - La liste des parametres
%% Return :La liste avec les paramètres réarrangé
%%-----
complete_a(List) ->
    Letter_list = lists:nth(1,List),
    Number_list = lists:nth(2,List),
    Number_list_average = lists:nth(3,List),
    [['letter list', Letter_list], ['number list', Number_list], ['number list average',
Number_list_average]].

%%% -----
%%% @version 0.1, {@date} {@time}.
%%% @author T. Pehme, L. Berthet
%%% @doc Le module {@module} presente la demarrage de l'application
%%% @end
%%% -----
-module(main).
-export([main/1]).

%%-----
%% La methode principale
%% Agra: PARAM: Le parametre pour le premier filtre
%%-----
main(PARAM) ->
    system:init(),
    system:start(PARAM).

```