Un service distribué sur le cloud pour l'exécution de chaînes de traitements sur la grille

Tristan Glatard
Université de Lyon, CREATIS ; CNRS UMR5220 ; Inserm U1044 ; INSA-Lyon ; Université Lyon 1, France
glatard@creatis.insa-lyon.fr

Overview

L'utilisation en production de plate-formes d'applications utilisant la grille demande des fonctionnalités toujours plus avancées concernant la description des applications [4], leur tolérance aux pannes [1] et l'optimisation de leur performance. Les gestionnaires de chaînes de traitements (*workflows*) en fournissent certaines, mais les services d'exécution associés doivent être distribués pour garantir le passage à l'échelle et réduire la dépendance à un point unique de défaillance. La distribution par services redondants, invoqués par *round-robin*, est très sensible aux pannes et nécessite une connaissance à priori des ressources et de leur caractéristiques. Nous proposons un système en *pool* auquel les gestionnaires de chaînes de traitements contribuent dynamiquement en fonction de leur performance et de leur état fonctionnel. Les gestionnaires sont déployés sur l'infrastructure *cloud* offerte par le projet StratusLab. Des expériences montrent la robustesse et le passage à l'échelle de l'approche proposée. A terme, une adaptation dynamique du nombre de gestionnaires en fonction de la charge de la plate-forme est envisageable. La mise en production du système dans la *Virtual Imaging Platform* (VIP [2]) est en cours.

Méthode et implémentation

La distribution par services redondants, invoqués par *round-robin*, est très sensible aux pannes. Considérons par exemple le cas d'une interface utilisateur de l'intergiciel `gLite`, sur laquelle 3 meta-ordonnanceurs (*Workload Management Systems - WMS*) sont configurés. Lorsque les 3 services sont fonctionnels, leur invocation en *round-robin* permet effectivement de distribuer la charge. En revanche, dès lors qu'un des services disfonctionne, un tiers des soumissions de tâches échoue faute de gestion appropriée des pannes. De même, l'invocation en *round-robin* ne permet pas de contrôler la charge imposée sur un *WMS* s'il vient à être surchargé. Ce problème, illustré ici sur l'exécution de tâches, est transposable à l'exécution de chaînes de traitements.

Pour améliorer la tolérance aux pannes et le contrôle de la charge des services, nous proposons l'architecture décrite sur la Figure 1 pour distribuer les services d'exécution de chaînes de traitements. Cette architecture est similaire à celle utilisée par les systèmes d'exécution de tâches par *pilot jobs*. Le système est composé d'un *pool* central, auquel les clients peuvent soumettre des chaînes de traitements, suivre leur état, et récupérer leurs résultats. Les chaînes de traitements et leur dépendances sont groupées dans des *bundles*, annotés sémantiquement comme décrit dans [5]. Des agents, aussi appelés *workflow executors*, sont distribués et peuvent se connecter au *pool*, récupérer des chaînes de traitement à exécuter, lancer des moteurs d'exécution de chaînes de traitements, envoyer l'état des chaînes de traitements au *pool* et au clients, et transférer les résultats au *pool*.

L'état des chaînes de traitements est maintenu à la fois par le *pool* et par l'agent, comme indiqué sur la Figure 2. Quand une chaîne de traitement est soumise, le *pool* lui assigne l'état PENDING. Le *pool* diffuse périodiquement un message contenant la liste des chaînes de traitements dans l'état PENDING pour que les nouveaux agents soient informés. Un délai d'expiration peut être associé à l'état PENDING. Il expire lorsqu'aucun agent n'est en mesure d'exécuter la chaîne de traitement, par exemple en cas de forte charge. La chaîne de traitement est alors mise dans l'état KILLED. Si des demandes d'exécution sont faites par les agents, alors le *pool* sélectionne un agent, lui envoie la chaîne de traitements, et la met en état SENDING. La chaîne de traitements est alors transférées à l'agent sélectionné, et mise dans l'état SENT, ou FAILED si le transfert échoue. Un délai d'expiration (T_{sent}) est démarré pour détecter les chaînes de traitements bloquées dans cet état. S'il expire avant que l'agent n'envoie un message d'état RUNNING, alors la chaîne de traitement est mise dans l'état KILLED. Lorsque la chaîne de traitements s'exécute, le *pool* attend des mises à jour d'états de la part de l'agent jusqu'à atteindre l'état FINISHED ou FAILED. La connexion avec l'agent est aussi testée périodiquement. Si elle est perdue, la chaîne de traitements est mise dans l'état KILLED après un délai d'expiration.

Lorsqu'un agent reçoit un message d'état PENDING, il vérifie le nombre de chaînes de traitements en cours d'exécution puis, si les conditions sont réunies, sélectionne aléatoirement une chaîne de traitement dans le message d'état, pour éviter les problèmes de concurrence entre les agents. La vérification du nombre de chaînes de traitements en cours d'exécution permet à l'agent de contrôler lui-même la charge qui lui est imposée. L'agent met alors la chaîne de traitement dans l'état WAITING, la demande au *pool* et démarre un délai d'expiration ($T_{waiting}$). Si ce délai expire, par exemple si le *pool* attribue l'exécution à un autre agent, alors l'exécution est supprimée. Sinon, la chaîne de traitements est mise dans l'état LAUNCHING et un moteur

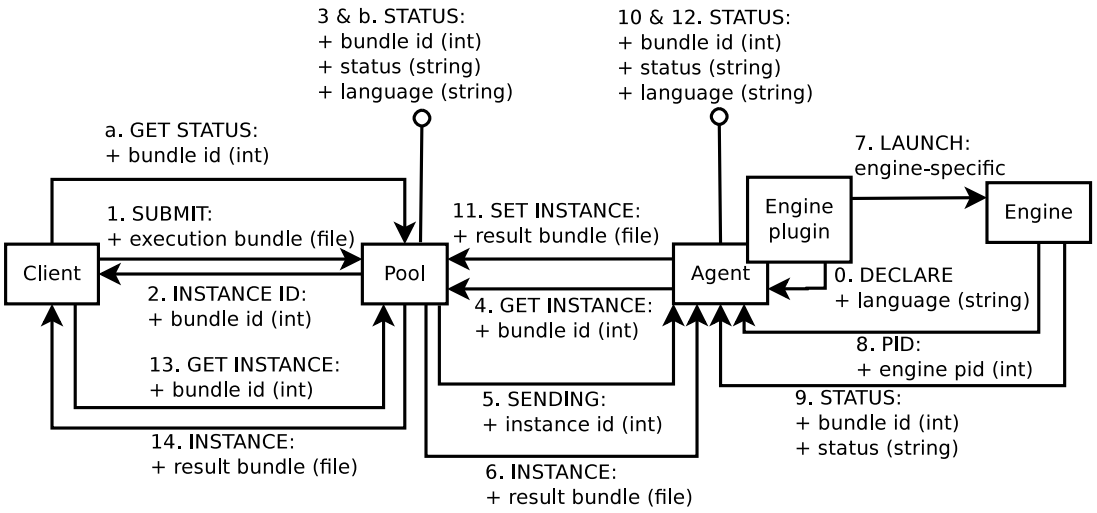


FIG. 1: Architecture du service distribué d'exécution de chaînes de traitements. La numérotation décrit les étapes de l'exécution d'une chaîne de traitements. Les flèches terminées par des cercles correspondent à des diffusions larges (*broadcast*.

d'exécution est déclenché. Un délai d'expiration ($T_{launching}$) est armé pour le cas où le moteur ne démarrerait pas. L'état de la chaîne de traitement est alors mis à jour jusqu'à sa complétion ou son échec. L'agent tue aussi les chaînes de traitements en cours d'exécution lorsque leur moteur ne répond plus après un délai d'expiration $T_{engine\_crashed}$.

Un prototype de *pool*, agent et client a été implémenté en Java. Le protocole XMPP (*Extensible Messaging and Presence Protocol*[1]) a été utilisé pour la couche de communication du fait de sa capacité à permettre la communication en environnement distribué sans nécessiter de connectivité entrante. Seul le serveur XMPP doit avoir un port ouvert. L'API Java smack[2] v3.2.2 a été utilisée. Le *pool* et l'agent ont chacun 3 *threads* pour recevoir et traiter les messages, pour transférer les fichiers, et pour suivre les délais d'expiration. Des bases de données assurent la persistance des états pour permettre au *pool* et à l'agent de redémarrer sans impact sur les chaînes de traitements actives. Une interface Java utilisant JSPF (*Java Simple Plugin Framework*[3]) est fournie pour permettre le développement des plugins. Elle a deux méthodes, pour lancer l'exécution des chaînes de traitements et récupérer leurs résultats. Un plugin agent a été développé pour les gestionnaires de chaînes de traitements MOTEUR [3] et Triana [6].

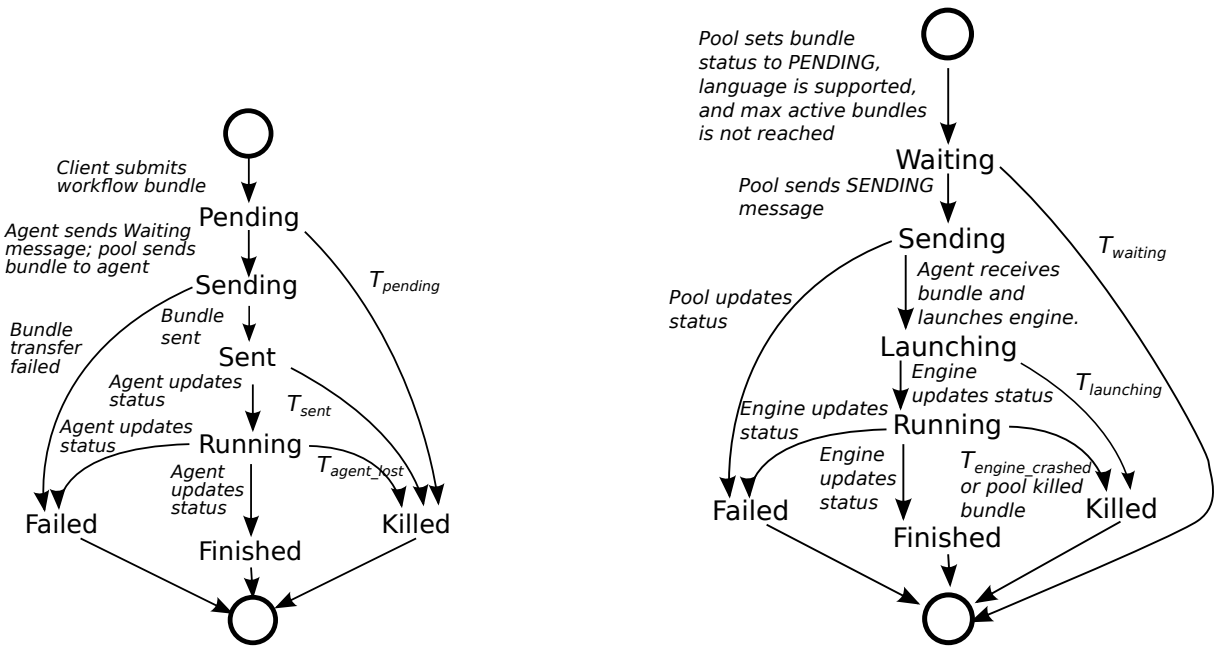
Expériences sur le cloud et résultats

Deux expériences sont rapportées ici pour étudier le passage à l'échelle et la robustesse de l'architecture proposée. La version 0.7 du *pool* de l'agent, du client et du plugin MOTEUR, disponible en ligne[4], est utilisée pour ces expériences.

Pour chacune de ces expériences, le *pool* et le clients sont déployés sur des machines différentes, sur le même réseau que le serveur XMPP. Les agents et les moteurs d'exécution sont déployés sur l'infrastructure *cloud* mise à disposition par le projet StratusLab[5]. Nous utilisons une machine virtuelle (VM) Fedora Core 16 x86_64 contenant Java, MySQL et notre agent. Des comptes XMPP sont créés manuellement pour les agents, et les noms d'utilisateurs et mots de passe sont configurés dans les VM déployées avant le démarrage des expériences. Les VMs sont déployées sur le site StratusLab du Laboratoire de l'Accélérateur Linéaire à Paris, avant le démarrage des expériences. Les délais d'expiration sont de T_{sent}=30 s, T_{agent_lost}=10 s, $T_{engine\_crashed}$=3 s, et $T_{waiting} = T_{launching}$=5 s. Le nombre maximum d'exécutions par agent est de 3. Le *pool* est configuré pour diffuser les messages d'état PENDING toutes les $\min(5s + n \times 0.1s, 300s)$, où n est le nombre de chaînes de traitements dans l'état PENDING.

Le passage à l'échelle est testé en terme de nombre d'exécutions de chaînes de traitements (`Exp1-a`) et de nombre d'agents (`Exp1-b`). Dans les deux cas, une chaîne de traitements MOTEUR constituée d'une seule activité dormant pendant 1 minute est utilisée. Les chaînes de traitements sont soumises séquentiellement au *pool*. Trois répétitions sont effectuées dans chacun des cas. Pour chaque répétition, le temps total de soumission et le *makespan* (durée entre le début de la soumission de la première chaîne de traitements et la fin de l'exécution de la dernière) sont mesurés.

[1] `http://xmpp.org`
[2] `http://www.igniterealtime.org/projects/smack/`
[3] `http://code.google.com/p/jspf`
[4] `http://vip.creatis.insa-lyon.fr:9002/projects/cgi-executor`
[5] `http://stratuslab.org`



(a) Dans le *pool* (b) Dans l'agent

FIG. 2: Automates d'état des chaînes de traitements. Les états initiaux et terminaux sont indiqués par des cercles. T indique un délai d'expiration (*timeout*).

	Test			Flapping			Crash	
	#Killed	Makespan(s)		#Killed	Makespan(s)		#Killed	Makespan(s)
#1	0	321	#1	0	319	#1	3	454
#2	0	318	#2	0	326	#2	3	385
#3	0	317	#3	0	319	#3	3	454

TAB. 1: Robustness of the execution pool to flapping and crashed agents.

Pour `Exp1-a`, 10 agents sont déployés, et le débit maximal est donc de 30 chaînes de traitements par minute (10 agents déployés, chacun pouvant exécuter 3 chaînes de traitement simultanément). Le nombre de chaînes de traitements exécutées varie de 10 à 150 par pas de 10. La Figure 3a montre l'évolution du *makespan* et du temps de soumission par rapport au nombre d'exécutions concurrentes. Les droites de régression aux moindres carrés sont aussi tracées. Le temps de soumission et le *makespan* sont tous les deux poches de leur droite de régression, ce qui indique le bon passage à l'échelle du système. La variabilité entre les répétitions est faible. Le temps de soumission est principalement contraint par le temps de transfert des chaînes de traitements (3.6 Ko), pénalisé par l'encodage *base 64* utilisé par XMPP pour les fichiers. La droite de régression du *makespan* a une pente inverse de 27,5 exécutions par minute, ce qui est proche du débit maximal sur l'infrastructure déployée.

Pour `Exp1-b`, le nombre d'exécutions est de 150 et le nombre d'agents varie de 1 à 10 par pas de 1. La Figure 3b montre l'évolution du temps de soumission et du facteur d'accélération en fonction du nombre d'agents déployés. Ce est calculé comme le rapport entre le temps cumulé d'exécution des chaînes de traitement (150 minutes) et le *makespan*. Comme attendu, le temps de soumission est stable. Les accélérations mesurées sont correctement approximées par leur droite de régression, ce qui indique que le surcoût du système reste contrôlé. La pente de la droite de régression est de 2,21 et l'accélération médiane pour 1 agent est de 2,6.

La fiabilité du système vis à vis des pannes survenant sur les agents est testée dans deux configurations : `flapping` et `crash`. Dans les deux cas, deux agents sont déployés. Dans la configuration `flapping`, la robustesse à la perte temporaire de connexion est testée. Un des agents se comporte normalement, et le second se déconnecte du *pool* pendant 5 secondes toutes les 10 secondes. Dans la configuration `crash`, les deux agents se comportent correctement pendant les premières 90 secondes, puis un agent se déconnecte jusqu'à la fin de l'expérience. Le *makespan* et le nombre d'exécutions échouées est mesuré dans les deux cas, et dans une configuration témoin où tous les agents se comportent normalement. La table 1 présente les résultats. Comme prévu, le système est totalement robuste à la configuration `flapping`. Le crash d'un agent n'a qu'un impact limité sur le système. Seules les chaînes de traitement en cours d'exécution au moment de la panne (3 dans notre cas) sont touchées, sans conséquence sur les exécutions ultérieures. Le *makespan* augmente par rapport à la configuration témoin du fait de la présence d'un seul agent après le crash.

Conclusion et perspectives

L'architecture proposée distribue l'exécution de chaînes de traitements entre plusieurs agents, par une approche similaire à celle utilisée par les *pilot jobs* pour la soumission de tâches. Le système résultant est robuste aux pannes des agents, et passe correctement à l'échelle. Les agents contrôlent eux-mêmes la charge qui leur est imposée. L'infrastructure *cloud* est appropriée au déploiement d'agents, ce que nous avons illustré en déployant nos agents sur un site de l'infrastructure StratusLab.

A court terme, une mise en production du système présenté ici dans la plate-forme VIP[6] est envisagée. Au delà des fonctionalités présentées plus haut, le *pool* permettra de gérer une file d'attente de chaînes de traitements en cas de saturation du système. A plus long terme, une étude du déploiement automatique d'agents sur le *cloud* en fonction de la charge de la plate-forme est envisagée. Le *cloud* serait en effet particulièrement adaptée à un déploiement élastique d'agents en fonction de la charge courante de la plate-forme.

Remerciements

Ce travail est financé par le projet SHIWA (numéro 261585), subventionné par l'accord de consortium 261585 de l'appel INFRASTRUCTURES-2010-2 du 7ème PCRD de la commission européenne. Nous remercions le projet StratusLab pour l'infrastructure *cloud* et le support utilisateurs.

Références

[1] R. Ferreira da Silva, T. Glatard, and F. Desprez. Self-healing of operational workflow incidents on distributed computing infrastructures. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing - CCGrid 2012*, pages 318–325, Ottawa, Canada, 05/2012 2012.

[2] T. Glatard, A. Marion, H. Benoit-Cattin, S. Camarasu-Pop, P. Clarysse, R. Ferreira da Silva, G. Forestier, B. Gibaud, C. Lartizien, H. Liebgott, K. Moulin, and D. Friboulet. Multi-modality image simulation with the virtual imaging platform : Illustration on cardiac mri and echography. In *IEEE International Symposium on Biomedical Imaging (ISBI)*, Barcelona, Spain, 2012.

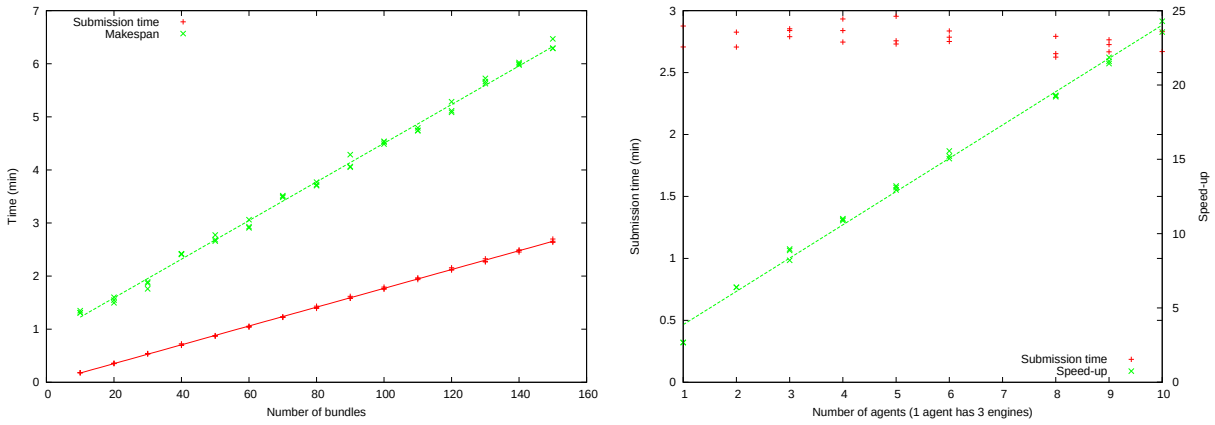
[3] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec. Flexible and efficient workflow deployment of data-intensive applications on grids with MOTEUR. *Int. J. High Perform. Comput. Appl.*, 22 :347–360, August 2008.

[4] K. Plankensteiner, J. Montagnat, and R. Prodan. IWIR : a language enabling portability across grid workflow systems. In *Proceedings of the 6th workshop on Workflows in support of large-scale science*, WORKS '11, pages 97–106, New York, NY, USA, 2011. ACM.

[5] I. Taylor, A. Harrison, D. Rogers, I. Harvey, and A. Jones. Object reuse and exchange for publishing and sharing workflows. In *Workshop on Workflows in Support of Large-Scale Science(WORKS'11)*, Seattle, USA, Nov. 2011.

[6] I. Taylor, M. Shields, I. Wang, and A. Harrison. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4) :153–169, September 2005.

[6] `http://vip.creatis.insa-lyon.fr`



(a) `Exp1-a` : en fonction du nombre d'exécution concurrentes (b) `Exp1-b` : en fonction du nombre d'agents

FIG. 3: Evaluation du passage à l'échelle de l'architecture en *pool*.