Evaluation et mise en place d’un serveur de messages pour Chamilo 2.0

Descriptif

Dans ce projet, on va montrer l’évaluation et la mise en place d’un serveur de message pour Chamilo. Le but est d’améliorer la montée en charge et les performances pour la partie « event » de Chamilo, c'est-à-dire les statistiques.

Le sous-système « statistique » de Dokeos consomme beaucoup de ressources et limite la montée en charge du serveur. Pour améliorer cette situation on implémente un serveur de message pour la partie statistique de son successeur : Chamilo. L’idée est la suivante, comme les statistiques n’ont pas besoins d’être traitées en temps réelle, on peut stocker les données dans une queue et les traiter de façon asynchrone, quand la charge du serveur est modérée. Ceci permet de libérer le serveur d’application et la base de données.

Solutions existantes

Apache ActiveMQ, bizTalk, etc.

Ici, on travaille en pure PHP, parce qu’on désire que tout le monde utilise le même format. Comme ca, on ne va pas avoir des problèmes avec l’interopérabilité entre différentes langages de programmation ou des environnements différents.

Description du Message server

Le serveur de message est une classe abstraite qui a les fonctions suivantes :

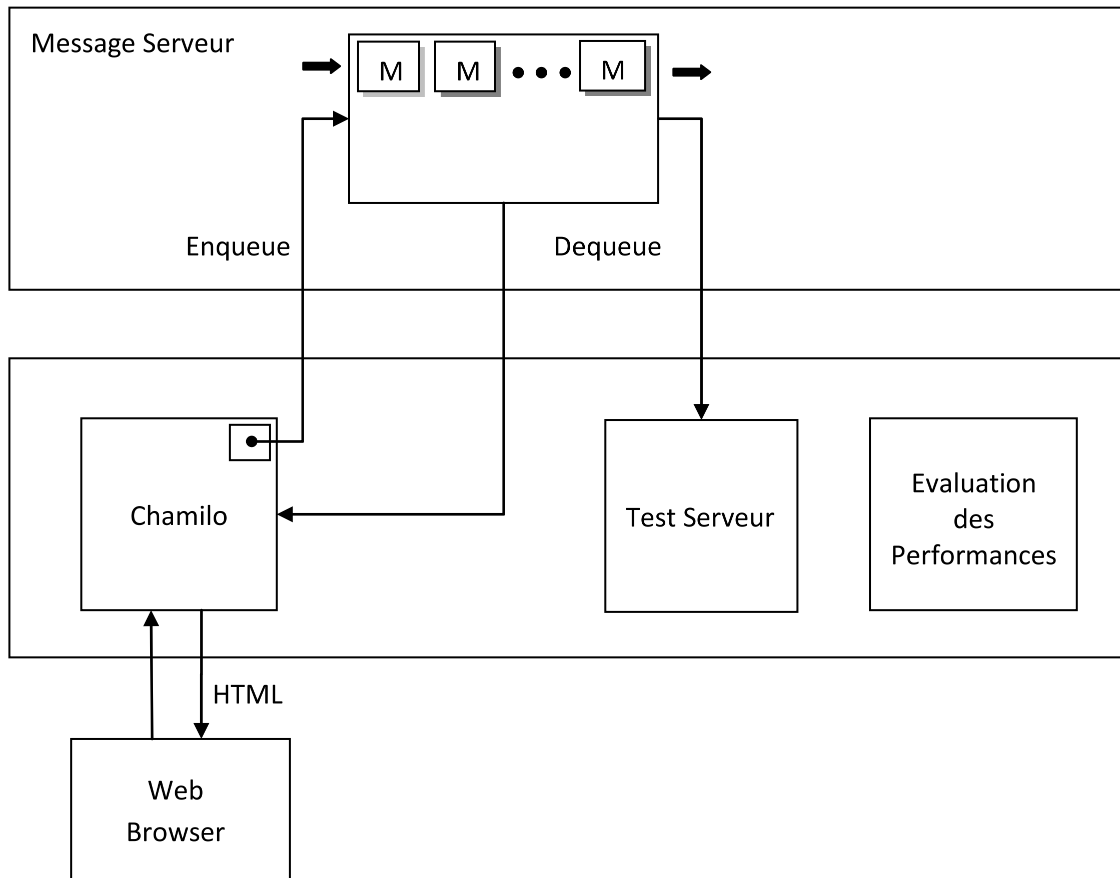
- get_instance
- enqueue(message)
- dequeue
- factory(method)

La fonction **get_instance** prend l’instance actuelle.

Le serveur est implémenté FIFO, on utilise **enqueue** et **dequeue**.

La fonction **factory** génère une nouvelle instance selon une certaine méthode.

Schéma



Les méthodes

J'ai plusieurs méthodes pour le serveur de message. Au début j'ai commencé avec une méthode et après j'ai cherché des autres méthodes pour améliorer le temps. Donc, il y a une évolution du temps.

1) Empty
2) Mémoire
3) Fichiers
4) Fichiers avec buffer
5) Fichiers avec APC

Chaque méthode étend la classe abstraite message server.

1) Empty

Le serveur qui ne fait que de retourner false.

2) Mémoire

Le serveur garde les messages dans la mémoire. On utilise les fonctions existantes dans PHP : array_push, array_pop.

3) Fichiers

Chaque événement est sérialisé et écrit dans un fichier de la forme suivante :

message_server_file_<TEMPS ACTUEL MICROTIME>_<# EVENEMENT>_<USER>.txt

La fonction dequeue tri les fichiers selon leurs noms, ainsi on peut extraire le fichier le plus ancien (paradigme FIFO), après on fait ‘unserialize’, on efface le fichier et on retourne le message.

4) Fichiers buffer

Un buffer de longueur fixe est utilisé. On ajoute dans le buffer chaque événement, aussi longtemps qu'il y a d'espace dans le buffer. Quand le buffer est complet, tous les événements sont écrits en fichiers et la procédure est reprise. A la fin, c'est important d'écrire les derniers événements dans le buffer, si le buffer n'est pas vide.

5) Fichiers avec APC

Le "Alternative PHP Cache" (APC) est un cache d'opcode libre et ouvert pour PHP. Son objectif est de fournir un framework libre, ouvert et robuste pour la mise en cache et l'optimisation de code intermédiaire PHP.

On utilise le cache pour un nombre fixe d'événements, après on passe chaque événement de cache en fichiers.

Détails de fonction

Il y a deux manières pour la création des événements.

1. Dans le browser, lorsque un utilisateur accède aux options du site (login, enter).
2. Les événements sont crées automatiquement (méthode de test).

On crée des événements, en mettant la méthode.

Ces événements sont détectés par la caractéristique de tracking du Chamilo qui fait appel au Message server pour enqueue le message.

Message server fait appel à factory pour utiliser la méthode qu'on désire. La factory reçoit la méthode comme paramètre, va sélecter la méthode parmi les méthodes existantes et va créer une nouvelle instance. Ca veut dire qu'elle va nous envoyer à la classe qu'implémente la méthode (instance).

Les messages sont enregistrés, en respectant la forme FIFO.

On utilise dequeue pour récupérer les messages.

Les testes de performance ont été faites pour toutes les méthodes, en utilisant 10, 100 et 1000 événements.

La classe **test_serveur** est définie pour tester les fonctionnalités.

Détails de performance

Le temps obtenu en utilisant les fichiers buffer et APC est montre une bonne performance. En suite, les autres méthodes indiquent un temps plus grand.

Le temps pour chaque exécution

	Empty	Mémoire	Fichiers	Fichiers Buffer	Fichiers APC	Default
10 messages	0.001214	0.001234	0.015067	0.001362	0.002115	0.397381
100 messages	0.011837	0.013200	0.165964	0.082123	0.082476	1.525664
1000 messages	0.130979	0.136307	1.410019	0.783486	0.807286	14.197292

Le rapport entre chaque méthode. Combien de fois une méthode est meilleure que l'autre ?

	Empty	Mémoire	Fichiers	Fichiers Buffer	Fichiers APC	Default
Empty	1	1.04067	12.411037	6.937822	6.967643	128.889414
Mémoire	0.983792	1	12.573030	6.221439	6.248181	115.580606
Fichiers	0.071322	0.079535	1	0.494824	0.496951	9.192740
Fichiers Buffer	0.144137	0.160734	2.020919	1	1.004298	18.577791
Fichiers APC	0.143520	0.160046	2.012270	0.995719	1	18.498278
Default	0.007758	0.008651	0.924200	0.513537	0.529137	1

Le serveur default c'est le point de départ. Le serveur qui utilise seulement la mémoire c'est la limite supérieure. Mais le problème pour le stockage de mémoire, c'est qu'après l'exécution les informations sont perdues.

Donc on désire d'avoir une méthode qui garde notre parcours, et qui a des bonnes performances.

Dans le premier cas, la méthode des fichiers, on observe que le temps c'est grand, parce que tous les événements sont stockes chacun dans un fichier.

Apres on a utilise une méthode avec le buffer qui redue considérablement le temps.

En suite, la méthode des fichiers en utilisant APC, c'est très bonne parce que APC fait l'optimisation du code PHP.

L'ordre de performance est la suivante :

default, fichiers, fichiers APC, fichiers buffer, mémoire, emtpy.

La classe **test_performances** est définie pour visualiser toutes les méthodes avec leurs temps d'exécution.

Observations

Observation 1

Pour le FIFO j'ai essaye d'utiliser posix_mkfifo, qui permets créer un fichier spécial FIFO qui existe dans le système de fichiers. J'ai rencontre des difficultés parce que cette librairie PHP s'utilise dans l'environnement UNIX.

Observation 2

Pour qu'APC marche, j'ai du installer le package PHP APC:

1) Dans php.ini on ajoute la ligne suivante
   extension = php_apc.dll
2) On ajoute les paramètres suivants
   apc.cache_by_default = On
   apc.enable_cli = Off
   apc.enabled = On
   apc.file_update_protection = 2
   apc.filters =
   apc.gc_ttl = 3600
   apc.include_once_override = Off
   apc.max_file_size = 1M
   apc.num_files_hint = 1000
   apc.optimization = Off
   apc.report_autofilter = Off
   apc.shm_segments = 1

```
apc.shm_size = 30
apc.slam_defense = 0
apc.stat = On
apc.ttl = 0
apc.user_entries_hint = 100
apc.user_ttl = 0
apc.write_lock = On
```

3) Le fichier php_apc.dll est copie dans le directeur php courant.
4) On teste avec une commande simple :

```
<?php print_r(apc_sma_info()); ?>
```

Observation 3

Pour l'utilisation du code la méthode factory a été utilisée. Si une autre méthode est ajoutée, il faut seulement modifier la factory. Dans cette manière le code peut être divisé sur des modules.

Les étapes pour l'intégration d'une nouvelle méthode :

- On crée la méthode, donc la classe php (ex : APC, buffer).
- On ajoute dans factory l'option pour la nouvelle méthode.
- Lorsqu'on crée des événements, il faut mettre la méthode comme paramètre.

Observation 4

J'ai utilise le Portfolio pour garder mon projet et d'avoir une trace de tout que j'ai travaille.