

PolyShare - Cloud computing

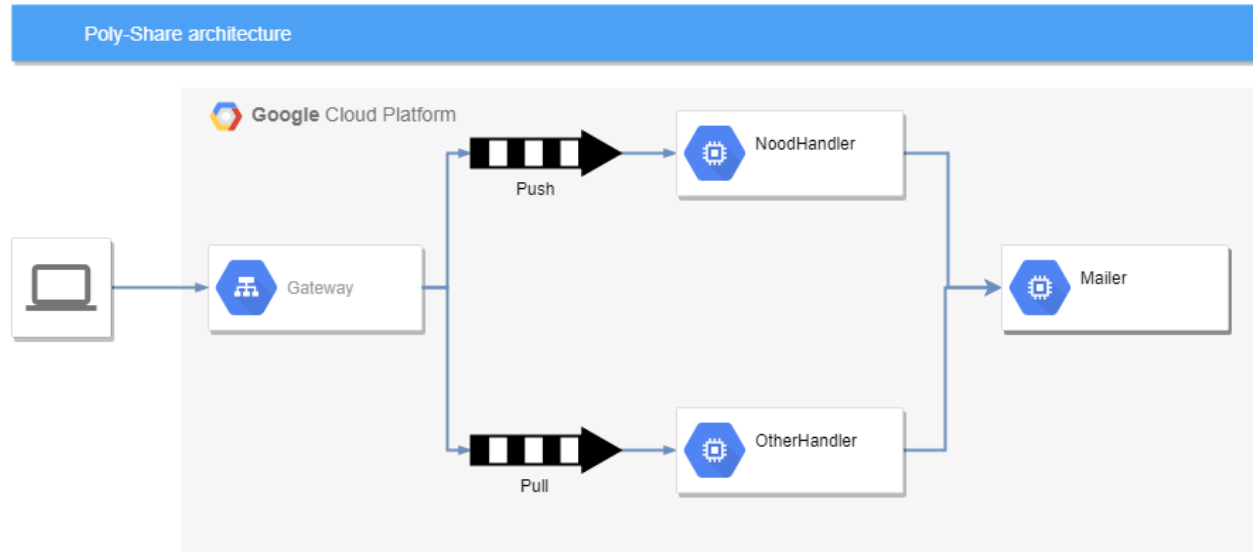
Rapport d'architecture

Alexandre CLÉMENT - Alexandre HILTCHER - David LANG - Florian LEHMANN



Ce document présente l'architecture globale de l'application Polyshare. Celle-ci est développée dans le cadre du cours de Cloud Computing qui utilise les services de Google Cloud Platform. Il présente l'architecture globale du projet et rentre ensuite dans les détails sur l'utilisation des queues de messages et l'élasticité. Finalement il présente une estimation du prix de l'application.

Architecture globale



Le diagramme ci-dessus présente l'architecture globale de notre application. Cette application est découpée en 4 composants et 2 queues de messages:

Le premier composant est la *Gateway*, il permet de rediriger les requêtes vers les différentes queues de messages. Il va aussi s'occuper de tout ce qui ne dépend pas des autres composants comme la génération de page HTML, la gestion des comptes utilisateurs.

Le *NoobHandler* va s'occuper de traiter les informations d'upload et de download des utilisateurs "*noob*". Il va donc pour chaque requête d'un utilisateur vérifier si cet utilisateur n'a pas dépassé son quota et ensuite transmettre un message au *Mailer*.

Le *OtherHandler* va s'occuper de gérer les deux autres types des comptes, en effet, ces derniers ne se verront pas refuser une requête si leur quota est dépassé, ils seront plutôt mis en attente. C'est pourquoi nous avons choisi de les séparer. De plus, ces deux composants n'utiliseront pas le même type de queue. En effet, si le *OtherHandler* a besoin d'une pull queue afin de pouvoir remettre en queue certains messages, le *NoobHandler*, n'en pas besoin.

Finalement, le *Mailer* est le composant responsable d'envoyer les mails aux différents utilisateurs afin de leur fournir les liens nécessaires à l'upload ou au download de leurs fichiers.

Utilisation des queues

Gestion des noobs

Les téléchargements des noobs vont être gérés grâce à une push queue. Les messages seront traités comme ils arrivent (FIFO). Pour vérifier si le noob peut ou non avoir le lien de téléchargement, le handler de la *push queue* va envoyer une requête à la base de données qui conserve l'état de chaque personne. Si la limite de téléchargement de la personne est atteinte alors la requête sera refusée.

La gestion de l'upload, se fait de la manière suivante : un utilisateur émet une requête qui va être mise dans la push queue. Lorsque le message est traité par le handler, si l'utilisateur n'a pas dépassé son quota, le handler va écrire les métadonnées dans la base de donnée et générer un lien permettant l'upload du fichier. Ensuite ce lien sera envoyé à l'utilisateur par mail afin qu'il puisse upload son fichier. Si l'utilisateur a dépassé son quota il recevra alors un mail avec le message suivant "*lol non noob*".

Gestion des autres

Les autres utilisateurs vont être gérés avec une pull queue. Si un utilisateur dépasse son quotas de téléchargements alors le message traité par le *handler* est remis dans la queue. Le gros problème de cette implémentation est de trouver un algorithme efficace pour dupliquer les handlers. En effet, nous devons faire la distinction entre une queue de message saturée car il y a trop d'utilisateurs différents ou une queue de message saturée car il y a trop de requêtes de mêmes utilisateurs.

L'upload sera géré de la même façon que pour les noobs. La différence principale est que cela sera géré par le *OtherHandler* via une pull queue. Ceci nous permet de ne pas être forcé à refuser la requête de l'utilisateur mais simplement le mettre en attente.

Utilisation de l'élasticité

Nous n'avons qu'un service, notre élasticité se fait donc au niveau de celui-ci. Quand la charge sera trop importante de nouvelles instances de celui-ci seront créées. Nous pourrions gérer plusieurs opérations en simultané grâce à la *Queue*. Il faudra néanmoins gérer la création d'instances des handlers de la pull queue comme expliqué dans la partie précédente, ce problème ne sera pas facile à résoudre.

En théorie nous aurions pu tout gérer avec une seule Queue afin d'en limiter le nombre et ainsi limiter le prix au maximum (si nous n'avons pas trop de messages stockés). Cependant, cette solution pourrait limiter la disponibilité de nos services à nos utilisateurs "privilégiés" car une seule Queue regrouperait l'ensemble des requêtes. Contrairement à la solution précédente, notre solution permet de fournir un accès privilégié à certains utilisateurs au dépit du passage à l'échelle qui est réalisé sur l'intégralité du service. Ce passage à l'échelle a lieu malgré le fait que nous puissions avoir un nombre asymétrique de message entre les deux queues de messages. Ce passage à l'échelle nécessite donc plus de ressources matérielles.

Coût de la plateforme

Le tableau ci-dessous présente une estimation des opérations effectuées pour chaque type d'utilisateur.

Type d'utilisateur	Nombre maximum d'opérations par minute	Nombre moyen d'opérations par jour	Nombre moyen d'opérations par mois
Noob	1	10	300
Casual	2	20	600
Leet	4	40	1 200

Nous estimons que chaque type d'utilisateurs (les noobs, les casuels et les leets) représentent à chaque instant (33%) des utilisateurs de la plateforme. Nous pouvons en déduire une moyenne de requête pour un utilisateur. Cette moyenne est donc de 23.3 opérations par jours par utilisateur.

A partir d'ici, afin de réaliser les calculs de coût, nous allons nous baser sur un pool de 1000 utilisateurs. Pour ce pool, nous avons donc en moyenne 23 000 opérations par jours à gérer.

Les quotas gratuits offerts par chaque service ne nous permettent pas de supporter le nombre d'utilisateurs définis précédemment. En supposant que chaque instance d'AppEngine supporte 100 utilisateurs, nous avons besoin de 10 instances par heures. Cependant, si on considère que la moitié du trafic se fera lors d'un pic d'utilisation, on peut donc dire qu'à ce moment-là on aura besoin de 5 instances. De ce fait, nous pouvons aussi considérer que pour le reste du temps nous aurons besoin de moins d'instances (disons 1). De plus, nous considérons que le pic d'activité se déroule entre 18 et 21h (quand les étudiants rentrent de cours). Nous avons donc 5 instances pendant 3h par jour et 1 pendant 21h. Ceci nous donne une moyenne de 1.125 instances par heure. Sachant que 2 instances par heure nous coûteraient \$71.5 par mois, le coût de notre application sera majoré par \$71.5.

Le service de Mailing de base de google cloud nous limite à un quota de 10 mails par jour ce qui est largement insuffisant lors d'une utilisation quotidienne par un grand nombre d'utilisateurs. C'est pourquoi nous envisageons d'utiliser le service Mailgun¹ qui permet d'envoyer plus de 5 millions de mails par mois. Nous avons estimé qu'il y a environ 2 fois plus de download que d'upload. À partir de là, nous pouvons estimer qu'il faudra envoyer environ 460 000 mails par mois. Le coût pour notre pool de 1000 utilisateurs s'élève donc à environ \$250 par mois.

Finalement, en prenant en compte le coût des instances et le coût des mails le prix mensuel total s'élèverait à environ \$321.5. Nos utilisateurs nous reviennent donc à \$0.315 par mois.

Type de stockage

Stockage des informations relatives aux utilisateurs

Afin de pouvoir limiter les utilisateurs sur l'upload, il est préférable que les informations soient cohérentes à tout instant. Nous avons donc choisi une base de données DataStore pour garantir la cohérence de la limite d'upload et du score de chaque utilisateur.

¹ <https://cloud.google.com/appengine/docs/standard/python/mail/mailgun>

Stockage des fichiers

Contrairement aux informations liées aux utilisateurs, les fichiers ne nécessitent pas de cohérences. En effet, les seuls risques qui peut y avoir sont :

- les accès après suppression. Les utilisateurs ont accès au fichier alors que celui-ci est censé être supprimé.
- les accès avant insertion. Les utilisateurs n'ont pas accès au fichier alors qu'il est censé être présent dans la base de données.

Ces deux risques ne sont pas forcément problématiques, nous avons décidé de les ignorer (on traite juste les erreurs).

La base de données choisie sera donc en NoSQL.

Coût de la couche de donnée

On utilisera pour le calcul du coût de la couche de donnée, les mêmes hypothèses que précédemment concernant le nombre d'utilisateurs utilisant notre plateforme. De plus, on suppose que les différents utilisateurs upload le nombre suivant de données par jour.

Groupe d'utilisateur	Noob	Casual	Leet
Données	30 Mo	100 Mo	250 Mo

Dans ce calcul, nous allons prendre en compte le stockage des données mais également le coût de la bande passante (trafic entrant et sortant). Nous supposons aussi que de la même manière que dans les parties précédentes, la moitié du trafic se fait sur un pic d'activité entre 18h et 21h. Ajoutons à cela que durant ce pic d'activité, l'utilisation est répartie de façon homogène.

Sur un pool de 1000 utilisateurs, on a donc 500 utilisateurs pendant le pic d'activité, ces utilisateurs sont répartis de manière égale sur les différents groupes.

On a donc environ 4 à 5 utilisateurs de chaque groupe qui vont upload des vidéos sur une période de 5 minutes (la durée maximum où une vidéo reste sur le serveur). Ces opérations génèrent environ 1.9 Go de donnée sur cette fenêtre de temps. Néanmoins, il faut prendre en compte que les fenêtres temporelles sont glissantes. En effet, les utilisateurs ne vont pas tous upload leurs vidéos exactement au même moment, il est donc possible de dépasser légèrement ce quota. Nous avons donc décidé d'allouer 2.5 Go de stockage, or comme nous utilisons le cloud storage pour stocker les fichiers et que les quotas gratuits sont de 5Go par mois, nous ne payons rien.

En plus de cela, il faut prendre en compte le nombre de lectures, écritures sur la base de données d'utilisateurs. Nous considérons que les Noob et les casual upload un seul fichier alors que les Leet upload jusqu'à deux fichiers. En prenant 250 personnes qui upload, dont un tiers qui upload 2 fichiers, nous obtenons 330 uploads par jour. Or la limite gratuite est de 20 000 écritures par jours, de même, les suppressions de fichiers ne seront pas facturées car le quota ne sera pas atteint. On ne paie donc rien en ce qui concerne l'upload de fichiers. En ce qui concerne le download de fichier, nous avons 750 utilisateurs qui download des fichiers chaque jour. En considérant qu'ils download 2 fichiers par personne et par jour on obtient 1500 lectures par jours. De la même manière que précédemment nous avons 50 000 écritures gratuites par jour, nous ne payons donc rien non plus.

Pour le calcul de la bande passante, elle n'est pas facturée pour la partie cloud storage. Cependant, nous allons considérer qu'il y a environ trois fois plus de downloads que d'uploads. Sur un pool de 1000 utilisateurs, on a donc 250 uploads et 750 downloads par utilisateurs. Or, le download de fichier ne nécessite pas d'envoi de requête vers la base de données DataStore. Il nous reste donc 250 requêtes, afin de modifier les statistiques des utilisateurs, sur la base de données. Soit approximativement 2 Kb par requête, on a un trafic entrant de 500Kb, nous seront donc facturé à hauteur de 1Go soit \$0.18 par mois.

Si on prend en compte le prix de toutes les composantes du système, on obtient un prix mensuel de \$0.18. Si on ajoute à cela le coût de la plateforme qui est de \$321.5 on obtient un prix total mensuel de \$321.68, soit environ \$0.32 par utilisateur et par mois.