


| | | |
|---|--------------------------|----------------------------|
|  | Dossier de conception | Version 1.0 9 août 2018 |
|---|--------------------------|----------------------------|

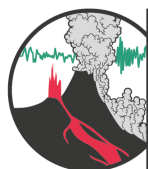
Institut de Physique du Globe de Paris

SiQaCo

Dossier de conception pour l'outil d'automatisation de la validation des données

| | |
|--------------|---|
| OBJET | Ce document contient le dossier de conception qui se base sur le cahier des charges fonctionnel de l'outil d'automatisation de la validation des données SiQaCo . |
|--------------|---|

| | |
|---------------|------------------|
| DATE | 21 novembre 2018 |
| STATUT | Première Version |



**Observatoires
volcanologiques
et sismologiques**
INSTITUT DE PHYSIQUE DU GLOBE DE PARIS



Olivier Geber, Arnaud Lemarchand, Constanza Pardo, Jean-Marie Saurel

Table des matières

| | |
|---|------------|
| Table des figures | iii |
| Documents de référence | 1 |
| 1 Introduction | 2 |
| 2 Architecture Générale | 3 |
| 2.1 Outils utilisés | 3 |
| 2.1.1 Environnements de développement | 3 |
| 2.1.2 Environnements de production | 3 |
| 2.1.3 Langages de programmation et bibliothèques utilisées | 3 |
| 2.1.4 Environnement de développement intégré (<i>IDE</i>) | 3 |
| 2.1.5 Gestionnaire de version | 4 |
| 2.2 Structure générale | 4 |
| 2.3 Utilisation | 4 |
| 3 Fonctionnalités | 5 |
| 3.1 Principales fonctionnalités du logiciel | 5 |
| 3.2 Fenêtre d'analyse et granularité | 5 |
| 3.3 Gestion des traitements | 6 |
| 3.4 Format des données | 7 |
| 3.5 Découpage des fichiers | 7 |
| 3.6 Journaux d'événements | 8 |
| 3.7 Historique des traitements | 8 |
| 3.8 Traitement des discontinuités | 8 |
| 3.8.1 Module d'analyse de l'archive finale | 8 |
| 3.8.2 Analyse de la disponibilité des sources | 9 |
| 3.8.3 Inventaire | 9 |
| 3.8.4 Requêtes | 9 |
| 3.8.5 Espace de travail | 10 |
| 3.8.6 Construction des requêtes | 10 |
| 3.8.7 Pile de requêtes | 11 |
| 3.8.8 Exécution des requêtes | 11 |
| 3.8.9 Traitement des requêtes | 12 |
| 3.9 Nettoyage des requêtes | 12 |
| 3.10 Modules de communication avec les sources | 13 |
| 3.10.1 Entrées et sorties de ces modules | 14 |
| 3.10.2 Disponibilité des données sur les sources | 14 |
| 3.10.3 Depuis le NAQS DataServer de Nanometrics | 14 |
| 3.10.4 Depuis le Web Service FDSN | 14 |
| 3.10.5 Depuis les numériseurs Nanometrics Centaur | 14 |
| 3.10.6 Depuis les numériseurs de Quanterra Q330 (Q330S, Q330 et Baler) | 15 |
| 3.10.7 Depuis les numériseurs Taurus de Nanometrics | 15 |
| 3.10.8 Depuis une station Guralp | 15 |
| 3.10.9 Depuis les numériseurs Géosig de Nanometrics | 15 |
| 3.10.10 Depuis une arborescence <i>offline</i> (récupération terrain sur clé USB, sur disque flash) | 15 |
| 3.11 Validation des données | 15 |

| | | |
|----------|---|-----------|
| 3.11.1 | Validation des métadonnées | 15 |
| 3.11.2 | Validation des données continues | 16 |
| 3.11.3 | Validation croisée des données et des métadonnées | 16 |
| 3.11.4 | Modification du label de qualité | 16 |
| 3.11.5 | Archive finale | 16 |
| 3.12 | Métriques | 16 |
| 3.13 | Développement de nouveaux modules | 17 |
| 3.14 | Classes | 17 |
| 4 | Interface Web | 18 |
| 4.1 | Configuration | 18 |
| 4.2 | Supervision | 18 |
| 4.2.1 | Contrôle passif | 18 |
| 4.2.2 | Intégration de l'outil Grafana | 18 |
| 4.2.3 | Visualisation des métriques | 19 |
| 4.2.4 | Visualisation de l'état de l'archive finale | 19 |
| 4.2.5 | Visualisation des tâches en cours | 20 |
| 4.2.6 | Visualisation des fichiers logs et des erreurs | 20 |
| 4.3 | Opérations manuelles | 20 |
| 4.3.1 | Récupération des données manuelles | 21 |
| 4.3.2 | Annuler ou relancer une ou plusieurs requêtes | 21 |
| 4.3.3 | Désactiver les alertes | 21 |
| 4.3.4 | Modifications manuelles de l'archive finale | 21 |
| 4.4 | Choix de l'environnement de développement | 21 |
| 4.5 | Lancement des scripts via un terminal | 23 |
| 5 | Bases de données | 24 |
| 5.1 | Outil de gestion de base de données | 24 |
| 5.2 | Gestion des utilisateurs | 24 |
| 5.3 | Tables | 24 |
| 6 | Planning du développement | 26 |
| 6.1 | Développement | 26 |
| 6.2 | Test | 26 |
| 6.3 | Production | 27 |
| 6.4 | Maintenance | 27 |
| | Appendix | 28 |
| A | Scripts existant de validation de données | 28 |
| A.1 | Validation des métadonnées | 28 |
| A.2 | Validation des données continues | 28 |
| B | Références | 29 |
| B.1 | Sites Web | 29 |
| C | Glossaire | 30 |
| C.1 | Noms | 30 |
| C.2 | Abréviations | 30 |

Table des figures

| | | |
|----|---|----|
| 1 | Schéma global du logiciel | 4 |
| 2 | Premier cas d'usage | 6 |
| 3 | Deuxième cas d'usage | 6 |
| 4 | Troisième cas d'usage | 6 |
| 5 | Structure générale de la solution | 7 |
| 6 | Analyse de l'archive finale | 8 |
| 7 | Principe de l'inventaire | 9 |
| 8 | Optimisation des requetes | 10 |
| 9 | Différents états des requêtes | 13 |
| 10 | Diagramme UML de SiQaCo | 17 |
| 11 | Visualisation de l'archive finale – <i>Source : ipgp.fr/ satriano/stats/index.html?year=2015</i> . | 19 |
| 12 | Visualisation d'une station – <i>Source : ipgp.fr/ satriano/stats/stats/GL</i> | 20 |
| 13 | Légende calendrier – <i>Source : ipgp.fr/ satriano/stats/index.html?year=2015</i> | 20 |
| 14 | Exemple d'un modèle – <i>Source : docs.djangoproject.com/fr/2.0/intro</i> | 22 |
| 15 | Exemple d'une vue – <i>Source : idem fig. 14</i> | 22 |
| 16 | Exemple d'un gabarit – <i>Source : idem fig. 14</i> | 23 |

Documents de référence

| Document | Version | Auteur.e(s) | Date |
|--|---------|---|------------|
| Cahier des charges fonctionnel SiQaCo (github.com/IPGP/siqaco) | 1.1 | Olivier Geber, Arnaud Lemarchand, Constanza Pardo, Jean-Marie Saurel | Avril 2018 |
| SEED Reference Manual (fdsn.org/seed_manual/SEEDManual_V2.4.pdf) | 2.4 | International Federation of Digital Seismograph Networks | Août 2012 |
| Documentation Django (docs.djangoproject.com/fr/2.1/) | 2.1 | Django Software Foundation | |
| Documentation Obspy (docs.obspy.org/) | 1.1.0 | ObsPy Development Team | |

Chapitre 1

Introduction

Ce document a pour but de présenter la solution technique au cahier des charges fonctionnel pour le développement du logiciel SiQaCo . Il se divisera en 5 parties :

- Description de la structure générale du logiciel (Chapitre 2)
- Description des différentes fonctionnalités (Chapitre 3)
- Description de l'interface Web pour la configuration du logiciel et les outils de visualisation (Chapitre 4)
- Description du gestionnaire de base de données (Chapitre 5)
- Planning du développement (Chapitre 6)

En guise d'introduction à ce document somme toute très technique, nous souhaitons rappeler que le logiciel qui sera développé a pour but d'être modulaire et collaboratif. Il sera donc documenté précisément afin que tout un chacun puisse le comprendre, et sera facile à l'installer et à utiliser. Enfin, le code sera rédigé de manière claire et compréhensible, en respectant les conventions d'écriture¹, et il sera commenté de manière exhaustive.

Les noms donnés aux classes, objets, méthodes, attributs etc. dans le code seront en anglais, ils seront donc également écrit en anglais dans le présent document.

1. Voir python.org/dev/peps/pep-0008

Chapitre 2

Architecture Générale

2.1 Outils utilisés

2.1.1 Environnements de développement

Le logiciel SiQaCo sera développé sur un environnement Linux, en l'occurrence sur la distribution **Ubuntu 18.04 (LTS)**¹.

Afin de faciliter l'installation des scripts Python et de la gestion de paquets nécessaire au fonctionnement des scripts, le logiciel **Anaconda** sera utilisé afin d'exporter l'environnement dans lequel les scripts fonctionnent. Ceci permettra en outre de ne pas avoir à modifier les fichiers système des machines des utilisateur.rices lors de l'installation du logiciel.

2.1.2 Environnements de production

Du fait de l'utilisation d'Anaconda, le logiciel SiQaCo pourra être utilisé depuis n'importe quel distribution Linux. Néanmoins, une attention particulière sera portée sur les environnements CentOS et Debian qui sont utilisés par les observatoires volcanologiques de l'IPGP (cf. Test 6.2) et par Géoscope.

2.1.3 Langages de programmation et librairies utilisées

La structure principale du programme sera développée en **Python 3.6**, et la manipulation de données sera effectuée à l'aide de la librairie **Obspy 1.1.0**.

Le programme nécessitera également de lancer des scripts **bash**, notamment pour changer d'environnement Python.

L'interface graphique sera une interface Web, développée avec l'environnement de développement (*framework*) **Django 2.0**.

La gestion de la base de données se fera grâce au SGBDR² **MySQL 14.14**, et avec l'ORM² de Django.

Ces choix seront expliqués au long de ce document.

2.1.4 Environnement de développement intégré (*IDE*)

Le développement du logiciel sera effectué avec Atom. Afin de faciliter le développement de scripts Python, les paquets suivants ont été installés :

— atom.io/packages/ide-python

1. Voir Annexe B pour les liens vers les différents environnements, *frameworks*, logiciels ...

2. Voir Annexe C pour le glossaire

- atom.io/packages/language-python
- atom.io/packages/script
- atom.io/packages/python-debugger

2.1.5 Gestionnaire de version

La gestion de version du logiciel SiQaCo sera faite avec git (2.17.0), et mise à disposition sur un dépôt Gitlab local. Lorsqu'il sera finalisé, le logiciel se trouvera sur un dépôt Github ou Gitlab. Pour l'instant, le dépôt Github se situe à l'adresse suivante github.com/IPGP/siqaco, mais il va très certainement migrer vers un dépôt Gitlab.

2.2 Structure générale

Le logiciel développé se décompose en 3 grandes parties interconnectées :

1. Le logiciel de traitement
2. L'interface Web
3. La base de données et ses différentes tables

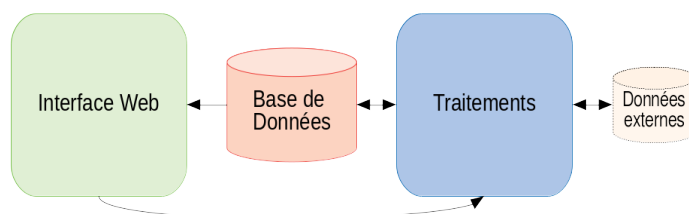


FIGURE 1 – Schéma global du logiciel

2.3 Utilisation

Le logiciel tournera comme tâche de fond (on pourra par exemple le configurer pour fonctionner comme un démon, lancé automatiquement au démarrage de la machine, ou selon une crontab, au choix).

Le logiciel a deux tâches principales qui sont : la construction d'une archive finale à partir de la lecture régulière de sources, et le contrôle qualité de cette archive.

Chacune de ces 2 tâche sera exécuté sur une certaine fenêtre temporelle d'analyse, avec un certaine granularité (voir plus loin) qui sera configuré par l'utilisateur.rice.

Lorsqu'il tourne en tâche de fond, l'utilisateur.rice pourra effectuer différentes opérations :

- Soit via un terminal, depuis lequel il pourra lancer des commandes propres au logiciel
- Soit via un navigateur Web, depuis lequel il pourra modifier la configuration du logiciel, intervenir sur les traitements en cours, visualiser l'état de l'archive et des traitements

Chapitre 3

Fonctionnalités

3.1 Principales fonctionnalités du logiciel

La fonctionnalité première de SiQaCo est de créer et de mettre à jour une structure finale de données et de méta-données à partir de différentes sources de données et de méta-données. La seconde fonctionnalité du logiciel sera d'effectuer le contrôle qualité sur les données et méta-données de l'archive finale créée.

3.2 Fenêtre d'analyse et granularité

La référence de temps qui sera utilisée dans SiQaCo sera la seconde. Néanmoins, l'utilisateur.rice pourra exprimer les durées en heures, jour, mois etc.

Soit T la période selon laquelle l'analyse sera lancée (ie. la granularité, définie en heures).

Soit D le délai de latence

Soit N la date courante

Soit F la durée de la fenêtre d'analyse (qui est dynamique : voir plus bas)

Le logiciel sera donc exécuté tout les T temps sur une fenêtre de temps de $t1$ à $t2$, où $t1$ est définie par

$$t1 = N - D - F$$

et $t2$ est définie par

$$t2 = N - D$$

proto

De plus, une variable *last_exec* gardera en mémoire l'heure de la fin de la dernière exécution. Ainsi, si l'exécution du traitement des discontinuités dure plus longtemps que la période de l'analyse, la taille de la fenêtre d'analyse devra être recalculée afin de ne pas perdre d'information.

Mathématiquement, cela se traduit par : si *last_exec* est antérieur à $N - D - F$, la fenêtre d'analyse devra commencer à :

$$t1 = last_exec$$

Exemple 1 : Analyse des 3 derniers jours toutes les 24 H, avec un délai d'1 jour

Dans ce cas, on définit $T = 1$ jour, $D = 1$ jour, $F = 3$ jours.

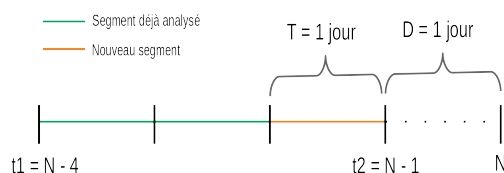


FIGURE 2 – Premier cas d'usage

Exemple 2 : Analyse des dernières 24H toutes les 2H, avec un délai d'1 heure

Dans ce cas, on définit $T = 2$ heures, $D = 24$ heures, $F = 24$ heures.



FIGURE 3 – Deuxième cas d'usage

Exemple 3 : Analyse d'une fenêtre de 15 jours commençant 30 jours avant la date courante, tous les jours

Dans ce cas, on définit $T = 1$ jour, $D = 15$ jours, $F = 15$ jours.

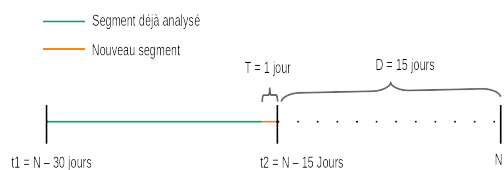


FIGURE 4 – Troisième cas d'usage

3.3 Gestion des traitements

D'après le cahier des charges fonctionnel de SiQaCo, voici la chaîne de traitement qui sera effectuée :

1. Dans un premier temps, le logiciel ira traiter les discontinuités depuis une ou plusieurs sources :
 - (a) Analyse de l'archive finale (métriques, liste des trous)
 - (b) Disponibilité des sources
 - (c) Inventaire de l'état des sources
 - (d) Construction d'une requête pour traiter les discontinuités
 - (e) Gestion des requêtes
 - (f) Exécution de la requête et mise à jour de la requête
 - (g) Traitement des discontinuités
2. Ensuite, le logiciel ira effectuer le contrôle qualité de l'archive finale. Celui-ci comprend :
 - (a) La vérification des métadonnées
 - (b) La vérification croisée des données avec les métadonnées
 - (c) La vérification des séries temporelles
 - (d) La modification du label de qualité

Voici le schéma de développement qui reprend l'idée générale du logiciel SiQaCo :

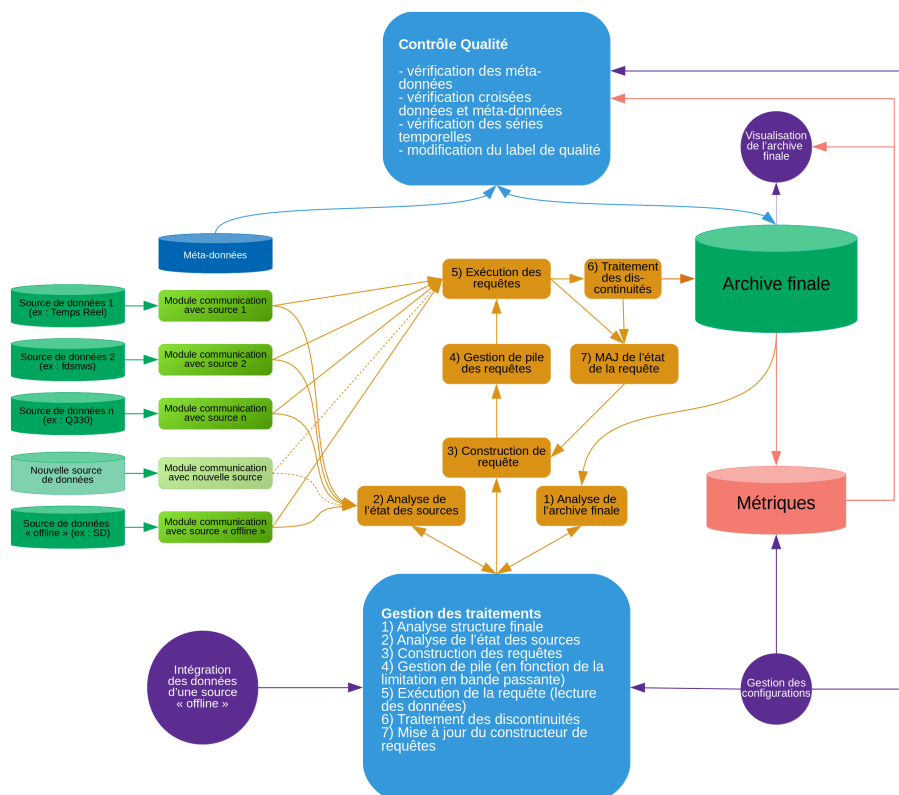


FIGURE 5 – Structure générale de la solution

3.4 Format des données

Le format de données que nous utiliserons, comme défini dans le cahier des charges, est le format MiniSEED, les métadonnées aux formats Dataless SEED et StationXML.

Ces formats étant voués à évoluer, voir à être abandonnés au profit d'autres formats, le logiciel devra également pouvoir évoluer pour pouvoir fonctionner avec d'autres formats.

Nous avons choisi d'utiliser la librairie Obspy qui prend en charge le miniseed, le Dataless SEED et le format station XML : l'ensemble du logiciel fonctionnera en utilisant les *stream* Obspy (qui sont définis dans la classe `obsypy.core.stream.Stream`¹), qui fonctionneront donc indifféremment du format d'entrée des données ou métadonnées. Les données qui seront écrites dans l'archive finale seront quand à elle copiées depuis les sources et bien qu'elles puissent être modifiées (redécoupées, compressée en 4ko etc.) leur format sera défini par les différents modules de lecture de données depuis les sources (cf. paragraphe 3.10).

3.5 Découpage des fichiers

Afin de satisfaire à la contrainte de découpage des fichiers de minuit à minuit énoncée dans le cahier des charges, un module se chargera d'effectuer ce dit découpage, en prenant garde à ne pas introduire de discontinuité, avec une attention particulière portée sur les secondes intercalaires (*leap seconds*).

1. docs.obspy.org/packages/autogen/obsypy.core.stream.Stream

3.6 Journaux d'événements

Le logiciel générera un journal d'événements (*fichiers "log"*) relatif à son fonctionnement, ainsi qu'un journal d'événements relatif aux traitements effectués, et nous ferons attention à ce que leur taille ne vienne pas poser de problèmes de ressources.

3.7 Historique des traitements

Le logiciel consignera également dans une table de la base de donnée l'ensemble des requêtes effectuées afin d'en obtenir un historique.

3.8 Traitement des discontinuités

Le logiciel SiQaCo est configuré pour lancer l'ordonnanceur de *traitement des discontinuités* sur une fenêtre de temps et une granularité définis par l'utilisateur. Les différentes étapes lancées par l'ordonnanceur ont pour but de venir peupler une archive dite *archive finale*, à partir de différentes sources lues par ordre de priorité, elles aussi configurées par l'utilisateur. Cette archive finale pourra contenir des trous mais jamais de recouvrements.

A chaque fois que l'ordonnanceur est exécuté, voici les différents modules qui sont appelés, dans l'ordre.

3.8.1 Module d'analyse de l'archive finale

Ce module ira lancer l'analyse des trous dans l'archive finale et ira remplir une table de l'état de l'archive finale. La liste des trous sera donnée par la méthode *get_gaps()* de *obspace.core.stream.Stream*.

Reprenons le cas d'usage 1 (cf. 3.2) :

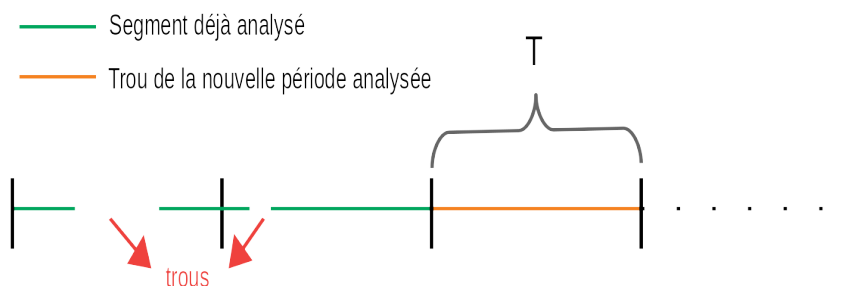


FIGURE 6 – Analyse de l'archive finale

Cette analyse va instancier deux objets : un objet *Gaps* et un objet *Overlaps* qui contiennent respectivement la liste des trous et des recouvrements présents sur l'archive finale. Chaque trou aura un statut (qui sera *new*, *in_process*, *done* ou *archived*). Lors de l'analyse, un nouveau trou recevra le statut *new*. Lorsqu'une requête (cf. paragraphe 3.8.4) sera créé pour ce trou, son statut passera à *in_process*, lorsque le trou est bouché ou que les données n'existent sur aucune source (cf. paragraphe 3.8.3) il passe à *done* et lorsque la date de fin du trou a dépassé la date de début d'analyse de l'archive finale ~~ET que son statut est *done*~~, son statut passe à *archived*. Cela nous permet de garder un historique des discontinuités qui ont été traitées.

3.8.2 Analyse de la disponibilité des sources

Ce module ira effectuer une analyse de disponibilité des sources *en ligne*, et renseignera une table de la base de données qui donnera la disponibilité des sources pour le traitement en cours. Si une source n'est pas disponible, le logiciel renverra une alerte à l'utilisateur.rice (cf. Visualisation des alertes au paragraphe 4.3 et Désactiver les alertes au paragraphe 4.3.3). Cette table sera effacée petit à petit, en fonction de la fenêtre d'analyse définie par l'utilisateur.rice, afin de ne pas générer de données inutiles.

Cependant, avant d'être effacée, les disponibilités des sources iront renseigner une seconde table de la base de données qui contient les statistiques de disponibilités de chaque source.

3.8.3 Inventaire

Chaque source aura un inventaire associé qui définit la disponibilité des données sur ladite source. Par défaut, l'inventaire définit les données comme étant disponibles en tout instant sur la source.

Si une source possède un outil qui permet de faire l'inventaire des données, cet outil ira mettre à jour cet inventaire (cf. paragraphe 3.10.2)

L'inventaire sera mis à jour continuellement : à chaque fois que l'ordonnanceur du traitement des discontinuités est exécuté (en faisant appel à l'outil d'inventaire des sources s'il existe), puis à la fin de l'exécution d'une requête : si en essayant de récupérer des données depuis une source dont les données sont définies comme disponibles mais que l'on ne récupère pas la totalité des données (c'est à dire que même après exécution et succès de la requête il reste un ou plusieurs trous), dans ce cas l'inventaire de cette source sera mis à jour afin de ne plus redemander de données qui n'existent pas sur cette source.

Cet inventaire pourra-être désactivé pour une ou plusieurs sources (dans la configuration du logiciel). En effet, dans certain cas, un trou présent à un instant t pourrait être comblé automatiquement à un instant futur $t + dt$.

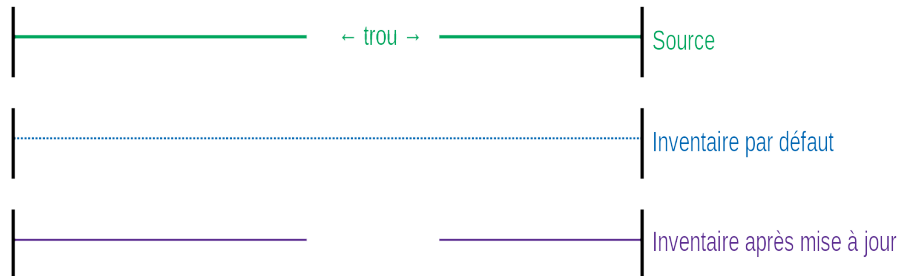


FIGURE 7 – Principe de l'inventaire

Lorsque ces trois premiers modules auront terminé leurs tâches respectives, le constructeur de requête va pouvoir commencer son travail.

Remarque : On pourra penser à paralléliser l'exécution de ces trois premiers modules puisqu'ils sont indépendants.

3.8.4 Requêtes

Une requête sera un objet de la classe *request*.

Les attributs des objets *request* sont les suivants :

- son statut : *status*
- sa durée de vie : *lifespan*
- la liste des sources avec leur priorité : *sources*
- la station, le réseau, le canal et le *LocID* : *sta*, *net*, *chan*, *loc*
- le début de la discontinuité à traiter : *starttime*

- la fin de la discontinuité à traiter : *endtime*
- un chemin vers un espace de travail : *workspace* (cf. paragraphe 3.8.5).

Les différents statuts des requêtes sont les suivants :

- *new* : c'est l'état (par défaut) d'une requête lorsqu'elle est créée puis ajoutée à la pile
- *in_progress* : c'est l'état d'une requête lorsqu'elle est en cours de traitement
- *succeeded* : c'est l'état d'une requête qui a été exécutée avec succès
- *failed* : c'est l'état d'une requête qui a échoué durant son exécution (et a généré une erreur)
- *retry* : c'est l'état d'une requête qui doit être exécuté à nouveau (après avoir échoué)
- *on_hold* : c'est l'état d'une requête dont la durée de vie a expiré
- *cancelled* : c'est l'état d'une requête qui a été annulée par l'utilisateur.rice

La durée de vie d'une requête pourra être définie par :

- un certain nombre de jours avant expiration
- un certain nombre de tentatives avant expiration

Les méthodes des objets *request* sont les suivantes :

- *execute()* qui permet à l'objet de s'exécuter lorsqu'il est appelé par la pile de requête

3.8.5 Espace de travail

Lorsqu'elle est exécutée, une requête va ajouter les données qu'elle lit sur les différentes sources à un espace de travail temporaire (*workspace*) qui sera fusionné à l'archive finale lorsque le traitement sera effectué. Chaque requête ayant un identifiant unique, l'espace temporaire créé sera lui aussi unique et identifié par cette même clé. C'est pourquoi l'espace de travail d'une requête fait partie de ses attributs.

Cet espace sera créé au sein d'un répertoire de travail temporaire dont la taille sera définie par l'utilisateur.rice. Chaque fois qu'une requête est effacée, l'espace temporaire associé est supprimé.

3.8.6 Construction des requêtes

Cette méthode va s'appeler *create_request*. Sa tâche est de remplir la pile avec des requêtes.

Dans un premier temps, ce module va lire la pile de requête afin de récupérer les requêtes (s'il y en a) présentes.

Il va ensuite aller lire la base de données de SiQaCo, et va lire la liste des discontinuités à traiter (les objets *Gaps* et *Overlaps*) et à partir de quelles sources (dans la table de l'état des sources et celle de la disponibilité des sources), en fonction de l'heure courante et des ressources disponibles.

Remarque : Ici, seules les discontinuités ayant un statut *new* et *in_process* doivent être traitées.

Un paramètre *max_requests* sera configuré par l'utilisateur.rice. Il correspond au nombre maximum de requêtes qui seront créées par canal et par période d'analyse : ainsi, s'il est défini comme étant égal à 3, si on a plus que trois discontinuités à traiter sur la période d'analyse alors on regroupe l'ensemble des discontinuités en une seule, qui commencera avec la date de début de discontinuité de la plus ancienne discontinuité et ira jusqu'à la date de fin de la discontinuité la plus récente.

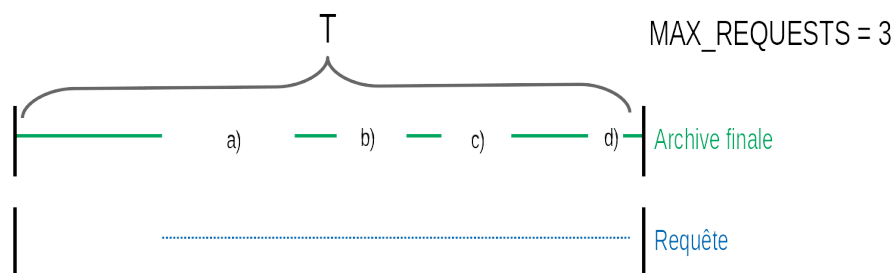


FIGURE 8 – Optimisation des requetes

Si une source n'est pas disponible à l'instant du traitement : elle ne sera pas prise en compte (la demande de données depuis cette source sera désactivée). Si une source est disponible, alors on vérifie qu'elle a des données disponibles sur la période qui nous intéresse, sinon elle ne sera pas prise en compte.

Le constructeur de requête va donc créer une requête par discontinuité (ou plusieurs si le paramètre `max_requests` est dépassé), et si cette discontinuité est la même pour chaque canal d'une station, voir sur plusieurs stations ou réseaux (ce dernier cas est supposé rare), alors la requête sera exécutée simultanément sur chacun des canaux/stations/réseaux.

Lorsqu'une requête est créée pour une discontinuité, son statut passe de *new* à *in_process*.

Si la requête en cours de création est fusionnée avec une ancienne requête (dans le cas où une requête était présente dans la pile avant d'entamer la création de requêtes, et qu'une nouvelle source est disponible pour la discontinuité en question), le statut de la discontinuité reste à *in_process*, (et la requête passe quand à elle de *retry* à *new*, voir 3.8.8).

Ces requêtes se baseront en outre sur la priorité des sources définies par l'utilisateur.rice lors de la configuration du logiciel. Ainsi, la requête sera construite de manière à ce que la source la plus prioritaire soit interrogée en premier, puis la seconde etc. Cela paraît trivial mais il vaut mieux être clair.

La requête sera alors envoyée dans la pile de requête en utilisant la méthode *add* de cette dernière, avec son état placé à *new*.

Lorsque toutes les discontinuités ont été traitées, le constructeur de requêtes aura terminé son travail.

3.8.7 Pile de requêtes

La gestion de pile se fera via l'utilisation des méthodes propres à la classe *list* de Python² Une pile contiendra une liste de requêtes dont l'ajout et la lecture dépendent de la façon dont la pile est définie par l'utilisateur.rice.

Dans le cas d'une pile FIFO³ (qui sera celle que nous implémenterons), la méthode implémentant l'écriture et la lecture sera définie de telle manière à ce que chaque nouvelle requête soit placée en "haut" de la pile et que la lecture de la pile renvoie la première requête de la pile.

Si par la suite un.e utilisateur.rice voudra développer une gestion de pile LIFO³, il suffira d'implémenter une nouvelle méthode mais où l'écriture et la lecture des requêtes sera définie de manière à ce que la dernière requête ajoutée soit la première à être retournée.

3.8.8 Exécution des requêtes

Cette méthode va aller lire des requêtes dans la pile et s'appellera *read_stack*. Seules les requêtes dont le statut est *new* ou *retry* seront exécutées.

Lorsqu'une requête est exécutée, son statut passe à l'état *in_progress*.

Afin que l'exécution d'une requête ne soit pas bloquante pour les autres, cette méthode devra être capable d'exécuter un certain nombre de requêtes de manière asynchrones⁴. Ainsi si une tâche est bloquée, les autres seront tout de même exécutées. Il sera donné à l'utilisateur.rice la possibilité de configurer combien de tâches il ou elle souhaite que le logiciel SiQaCo effectue parallèlement (et nous choisirons un nombre de tâche par défaut de 5).

Lorsqu'une requête est lue, elle va alors exécuter sa méthode *execute* qui ira lire les données depuis la ou les sources et les écrire dans l'espace de travail temporaire qui lui est associé.

Lorsqu'un ou plusieurs tâche(s) est/sont bloquée(s), une alerte est levée. (cf. ci-dessous)

2. docs.python.org/3/tutorial/datastructures

3. Voir Annexe C

4. Voir le module *threading* de Python 3 : docs.python.org/3/library/threading.html

3.8.9 Traitement des requêtes

Lorsqu'elle est exécutée, la requête va lire les données de la première source grâce au module correspondant à la source en question (cf. paragraphe 3.10).

Dans le cas d'un ou plusieurs trou(s) : les données sont copiées dans un espace de travail temporaire *workspace*, en leur donnant le nom correspondant à la structure de fichier définie par l'utilisateur.rice (un nom SDS par exemple) auquel sera ajouté un suffixe précisant qu'il s'agit de la source 1 (.S1 par exemple), et la taille des blocs ainsi que l'encodage seront également modifiés afin de correspondre à la configuration du logiciel. Cela se fera en modifiant le moins possible les blocs *miniseed* afin de ne pas introduire d'erreurs, en évitant au maximum les décompressions/compressions et portant un soin particulier à ne pas introduire de recouvrement, en particulier lors de *leap second*. Si le(s) trou(s) est/sont totalement rempli(s), la requête est alors terminée, le statut de la discontinuité passe à *done*.

Lorsque les données sont lues depuis la source, si plusieurs réseaux/stations/canaux sont demandés alors le logiciel SiQaCo vérifiera que l'ensemble des canaux a bien été lu. Si ce n'est pas le cas, une alerte est levée.

S'il reste des trous pour la période définie dans la requête à l'issue de ce traitement, ou qu'il manque des données sur un ou plusieurs canal/canaux, alors la requête ira lire les données de la seconde source dans l'ordre des priorités (si elle existe) qui seront copiés vers un fichier correspondant à la structure de fichier avec un suffixe précisant qu'il s'agit de la source 2 (.S2 par exemple), avec la bonne taille de bloc et le bon encodage, puis fusionner ces données avec celles qui ont été copiées précédemment, et cela jusqu'à épuisement des sources. Lorsqu'il n'y a plus de source à interroger, on passe à l'analyse des recouvrements sur les données que l'on a copié et fusionné dans l'espace de travail temporaire.

Dans le cas d'un ou plusieurs recouvrement(s), il nous faut traiter 4 éventualités :

1. En cas de *leap second*, si on voit un recouvrement d'une seconde le jour précédent ou suivant cette *leap second*, il faudra laisser la donnée.
2. Selon la configuration, on vérifie si ce recouvrement n'existe pas dans la source prioritaire. Si c'est le cas, on remplace la période de temps de l'archive finale par les données de cette source prioritaire.
3. Si ce recouvrement existe également dans la source prioritaire, le logiciel vérifie si ce recouvrement est deux fois la même donnée sur la même période de temps, dans ce cas il suffit de supprimer la moitié des données.
4. Enfin si les deux données qui se recouvrent sont différentes sur un même intervalle de temps, on crée un trou.

Lorsque ces traitements sont effectués, l'espace de travail *workspace* est fusionnée à l'archive finale.

- Si ces étapes se déroulent correctement, alors le statut de la requête passe à *succeeded*, et l'espace de travail temporaire est supprimé
- Si l'une de ces étapes échoue, alors le statut de la requête passe à *failed*, si sa durée de vie est définie par un nombre de tentatives, celle-ci est décrémentée, l'espace temporaire reste en l'état, et une alarme est levée
- Si l'utilisateur.rice choisit d'annuler une requête, alors le statut de la requête passe à *cancelled* et l'espace de travail temporaire est supprimé

3.9 Nettoyage des requêtes

A la fin de son traitement, une requête est analysée par une méthode de nettoyage de la pile que l'on nommera *clean_stack*.

Ce module va lire l'état de la requête en sortie de traitement.

1. Si son statut est *succeeded* ou *cancelled*, alors la requête est ajoutée à l'historique des requêtes puis est effacée.
2. Si son statut est *new*, *in_progress* ou *on_hold*, l'exécution de la requête est anormale : une erreur est levée, c'est à l'utilisateur.rice de choisir s'il veut la relancer (auquel cas son statut passe à *on_hold*) ou l'annuler (auquel cas son statut passe à *cancelled*).

3. Si son statut est *failed*, alors il y a plusieurs possibilités :

- Si sa durée de vie n'est pas expirée, alors son statut est placé à *retry*, sa durée de vie décrétementée (si elle est définie par un nombre de tentatives) puis est renvoyée à la pile de requête (en utilisant la méthode *add*)
- Si sa durée de vie est expirée, son statut est alors placé à *on_hold*, et la requête est renvoyée dans la pile de requête (mais ne sera plus lue et ne pourra plus être traité que manuellement cf. paragraphe 4.3.2).

Voici un schéma récapitulatif des différents états des requêtes :

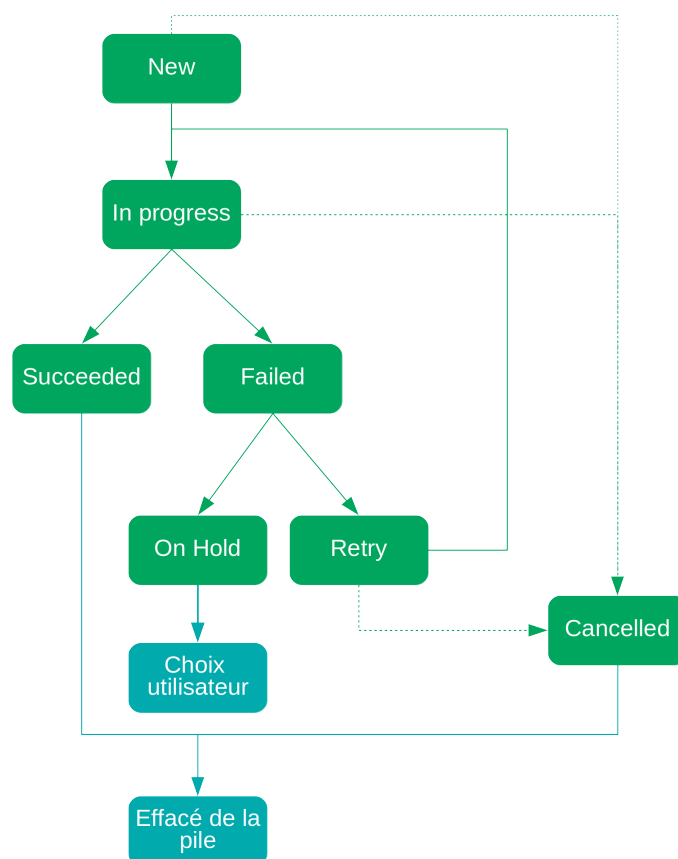


FIGURE 9 – Différents états des requêtes

En résumé, après avoir été traitée par ce module, la pile pourra être dans deux états :

- Vide
- Peuplée avec une ou plusieurs requête(s) dont le statut est *retry* ou *on_hold*

3.10 Modules de communication avec les sources

Chaque source aura un module qui permettra de communiquer avec elle. Ces modules auront deux fonctions principales : la première sera de lire l'inventaire des données disponibles pour une certaine période (qui correspond à la fenêtre d'analyse des sources définie par l'utilisateur.rice, cf. 2.3). Dans le cas où il n'existe pas de fonction qui permette d'accéder à l'inventaire des données présentes sur la source, les données sont définies comme étant disponibles sur l'ensemble de la fenêtre de temps analysée.

La seconde fonction de ces module sera de récupérer les données de cette source sur une période de temps définie.

3.10.1 Entrées et sorties de ces modules

Les modules de communication avec les sources devront pouvoir :

Transformer les données disponibles en *stream* Obspy, ainsi que de copier les données vers une archive temporaire de travail (le fameux *workspace*).

De plus, chaque source aura son inventaire (cf. paragraphe 3.8.3), où toutes les données sont disponibles par défaut lors de son initialisation.

3.10.2 Disponibilité des données sur les sources

Il ne s'agit pas ici d'un module à proprement parler, chaque source procèdera de manière différente. Ainsi, chaque module qui permettra de communiquer avec une source aura également une fonction qui permettra, si possible en fonction des sources, de savoir quelles sont les données disponibles sur sa source respective. Pour plus de détail, se référer aux modules des sources 3.10.

Cette fonctionnalité ira donc effectuer une analyse de disponibilité et des données sur les sources **sans récupérer les données** et renseignera une table de la base de données qui comprendra les périodes où les données y sont disponibles par canal. Cette table sera mise à jour à chaque exécution du logiciel.

3.10.3 Depuis le NAQS DataServer de Nanometrics

Les numériseurs de Nanometrics numérisent directement en SEED. La transmission des données se fait point à point par VSAT entre la station et l'observatoire. Le temps réel est optimisé, avec multiplexage en temps réel. Chaque station a un temps d'émission défini : un certain temps qui permet d'envoyer la donnée en temps réel + 10 à 20% de ce temps alloué à la retransmission des données.

Le script `nptonmnp` permet de transformer les données de NP (Nanometrics Protocol) en NMXP qui est compatible avec le NAQS. Le script `nmxp2tool` est un module `seiscomp` qui permet d'obtenir les données en temps réels ou celles du DataServer.

Dans le ringbuffer, on a 1 fichier par canal. Le buffer a une taille définie en octets, donc sa durée dépend du taux de compression, du bruit... Un canal correspond à 500Mo dans le buffer, ce qui correspond à peu près à 30 jours.

Selon la bande passante et la taille du trou, la demande de retransmission peut partir immédiatement ou prendre 1 à 2 journées à être envoyée vers la station.

En pratique, lorsque l'on redemande une donnée à la station, elle vient boucher le trou du ringbuffer (et pas de l'archive SDS).

Dans le cas du NAQS, il existe des commandes pour vérifier rapidement la liste des trous dans le NAQS, on pourra l'utiliser afin d'écrire l'état des données (cf. paragraphe 3.10.2) dans la table correspondante.

Il faut également pouvoir configurer la méthode de demande de données la plus économe en ressources entre la demande par internet des données à l'autre observatoire (si elle y est présente), ou bien la demande en local au DataServer.

3.10.4 Depuis le Web Service FDSN

à compléter

3.10.5 Depuis les numériseurs Nanometrics Centaur

à compléter

3.10.6 Depuis les numériseurs de Quanterra Q330 (Q330S, Q330 et Baler)

La récupération des données se fait depuis le Baler via une requête *ftp*
à compléter

3.10.7 Depuis les numériseurs Taurus de Nanometrics

à compléter

3.10.8 Depuis une station Guralp

La récupération des données se fait en *scp* ou via *rsync*

3.10.9 Depuis les numériseurs Géosig de Nanometrics

vérifier si ce développement sera vraiment fait ?

3.10.10 Depuis une arborescence *offline* (récupération terrain sur clé USB, sur disque flash)

L'outil devra pouvoir récupérer les fichiers présents dans une archive SDS de station.

La récupération peut se faire en transfert ftp depuis le serveur distant. Il faut préciser l'IP du serveur ainsi que le répertoire des fichiers. à compléter

3.11 Validation des données

Lorsque l'ordonnanceur de traitement des discontinuités a terminé son travail, l'ordonnanceur effectuant la validation des données sur l'archive finale est lancé. Nous l'appellerons *QualityControl*.

3.11.1 Validation des métadonnées

La validation des métadonnées diffère en fonction du format des métadonnées : dans les cas présent, on considère qu'elles seront soit au format StationXML (qui est le nouveau standard pour les métadonnées), soit au format dataless SEED.

Dans les cas des dataless SEED, la validation des métadonnées se fait pour l'instant selon plusieurs étapes, chacune correspondant à un script existant (cf. Appendix A.1)

Ces différents scripts seront repris, inclus au logiciel SiQaCo et automatisés dans l'outil développé. Certains scripts étant redondants avec le traitement des discontinuités effectué précédemment, il sera nécessaire de repenser ces scripts, et très certainement de les réécrire (avec Obspy plutôt qu'en Bash si possible). Pour la localisation des stations, un outil de cartographie intégré pourra faire l'affaire, plutôt que de lancer Google Earth⁵.

Pour rappel, voici les vérifications définies dans le cahier des charges :

- Conformité aux standards (Dataless SEED ou StationXML)
- Qu'il n'y a pas de recouvrement dans les époques des méta-données
- Que tous les canaux définis dans SiQaCo sont présents dans la méta-donnée
- Cohérence des dates dans les méta-données (canaux, stations, réseaux, commentaires)
- Vérification de la cohérence avec le nom du canal (fréquence d'échantillonnage, dip et l'azimut)

5. Voir par exemple le paquet Django django-location-field : github.com/caioariede/django-location-field

3.11.2 Validation des données continues

Les scripts de validation de données continues serviront de re-vérification des données (voir Appendix A.2). Pour rappel, voici les vérifications définies dans le cahier des charges :

- Vérifier que les “NSLC” Network/Station/LocID/Channel y soient correctement renseignés
- Vérifier la cohérence du nom et de l’arborescence
- Vérifier que le label de qualité soit celui attendu
- Vérifier qu’il n’y ait pas d’erreur lors de la décompression de la donnée
- Vérifier que la fréquence d’échantillonnage est homogène
- Vérifier que l’encodage est homogène et conforme à celui attendu
- Vérifier que la taille de blocs est conforme à celle attendue et homogène
- Vérifier que l’ordre des octets soit correct (en big endian)
- Vérifier les recouvrements qui sont toujours présents dans les données

3.11.3 Validation croisée des données et des métadonnées

Cette étape consiste à vérifier la cohérence des informations partagées entre les méta-données et les séries temporelles :

- Vérification de la cohérence d’informations redondantes tel que le type de compression et de la taille des blocs
- Vérification de la cohérence dans les réponses instrumentales
- Vérification que tout échantillon ait une méta-donnée unique

Ces trois vérifications devront pouvoir s’effectuer indépendamment du format de données ou du type de compression attendu en sortie de notre traitement. Ces paramètres seront définis par l’utilisateur.rice.

3.11.4 Modification du label de qualité

A la fin des 3 premières étapes, le logiciel ira modifier le label de qualité sur les données validées par le processus de contrôle de qualité.

3.11.5 Archive finale

Après la validation des données automatique, l’archive finale devra comporter des fichiers découpés de minuit à minuit (qu’il y ait eu traitement des discontinuités ou non).

Un module spécifique de traitement des fichiers s’occupera de découper les fichiers présents sur l’archive finale, à l’issu du processus de contrôle qualité. Il s’exécutera s’il voit qu’une journée entière vient d’être validée.

Le format des données et le type de compression seront défini par l’utilisateur.rice. Les données seront par défaut au format miniseed et dataless SEED, compressés en Steim2 à 4096o (cf. paragraphe 3.8.9).

3.12 Métriques

SiQaCo permettra de lancer un outil d’analyse de métriques sur l’archive finale. Cet outil sera choisi par l’utilisateur, et le logiciel lancera le module correspondant à l’outil en question. Dans le cadre de notre développement, nous avons choisi d’utiliser le logiciel ISPAQ (cf. Appendice : B). Cependant, l’utilisateur.rice pourra ensuite développer son propre module (cf. paragraphe 3.13) afin d’utiliser d’autres outils, comme par exemple WaveFormCatalog (cf. Appendice B).

Les modules permettant l’analyse des métriques généreront un document .csv standardisé qui seront lus par SiQaCo et directement intégrés à la table des métriques de la base de données de SiQaCo .

Lors du développement du logiciel, nous nous intéresserons aux métriques suivante : valeurs moyenne, médiane, minimale et maximale du signal, détection et nombre de pics dans le signal, RMS (root mean

square), nombre de trous et de recouvrements et le pourcentage de disponibilité des données.

La configuration du logiciel permettra de définir la ou les périodes sur laquelle/lesquelles on veut avoir les métriques. On pourra ainsi avoir, par exemple, une métrique par jour, ou une par mois, par an...

Cette analyse sera configurée depuis l'interface Web, qui permettra en outre la visualisation de l'état de l'archive (mise à jour à chaque execution du logiciel).

3.13 Développement de nouveaux modules

Le logiciel SiQaCo étant développé de manière modulaire, il sera donné une importance particulière à la lisibilité des différents modules afin de pouvoir par la suite développer de nouveaux modules. Par exemple, dans le cas des métriques, le logiciel SiQaCo sera configuré par défaut pour utiliser l'outil ISPAQ, et ira lire le script (*Bash* en l'occurrence) qui lancera l'outil ISPAQ sur l'archive finale. Lorsqu'un.e utilisateur.ice voudra développer un nouveau module afin d'utiliser un autre outil (par exemple WaveFormCatalog, cf Appendice B), l'intégration de ce nouveau module devra être évidente. C'est à dire que la façon d'écrire ce module, de le placer au sein du code déjà existant, comment le code ira l'appeler et comment le tester devront être suffisamment bien documenté afin d'éviter de rendre le programme "illisible" au cours de futurs développements.

3.14 Classes

Voici le diagramme UML des classes du logiciel SiQaCo . Il se peut que ce diagramme évolue au fur et à mesure du développement, le diagramme suivant nous donne une idée générale des classes que nous définirons :

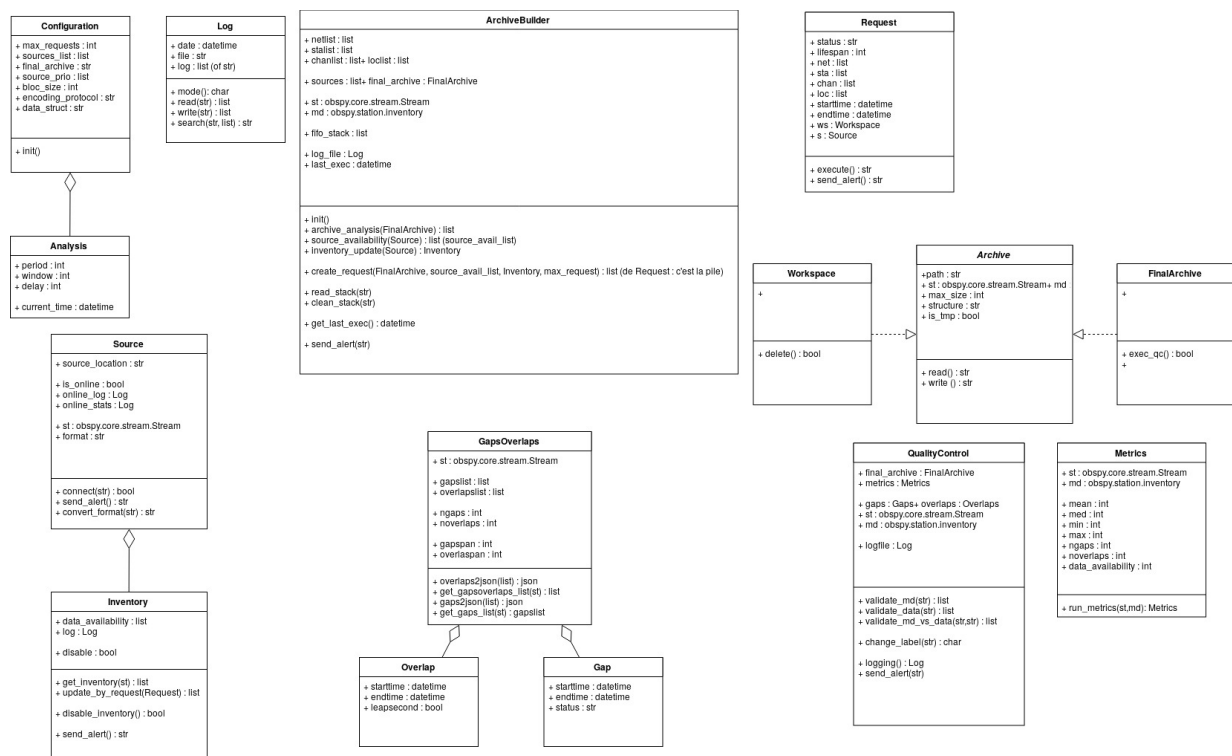


FIGURE 10 – Diagramme UML de SiQaCo

Chapitre 4

Interface Web

4.1 Configuration

La première fonctionnalité de l'interface web sera de configurer le logiciel. Nous rappelons que le logiciel devra également être paramétrable en ligne de commandes (cf. ci-dessus). L'idée principale est que la configuration du logiciel donnera le choix à l'utilisateur.rice du module à utiliser pour effectuer la chaîne d'actions qui va permettre de traiter les discontinuités et d'effectuer le contrôle qualité de l'archive finale.

Pour chaque station et chaque canal, le logiciel va permettre de configurer les paramètres suivants :

- Format de données et méta-données (miniSEED, stationXML)
- Type d'architecture (SDS, BUD...)
- Sources (Baler, SD, Cygnus...) à scruter
- Période d'analyse des sources (durée de temps analysée)
- Hiérarchie des sources
- Liste des canaux attendus
- Fréquence d'exécution du logiciel (c'est à dire la fréquence où les sources sont interrogées)
- Durée de vie des requêtes N ou dT par source (et choisir si nombre d'essai ou delta T)
- Limitations en bande passante manuelle
- Choix du format de données de sortie souhaité (quel compression, quel label de qualité...)
- Nombre de requêtes exécutées parallèlement

L'utilisateur.rice pourra, via cette interface, sauvegarder une configuration pour une station/un canal et l'utiliser pour d'autres canaux.

La modification de la configuration ne pourra se faire qu'en l'absence de travail en cours, et prendra effet dès qu'elle est validée.

4.2 Supervision

4.2.1 Contrôle passif

SiQaCo devra générer des commandes externes (en suivant le protocole SNMP par exemple) pour le logiciel de supervision NAGIOS : (wiki.monitoring-fr.org/nagios/addons/nsca)

4.2.2 Intégration de l'outil Grafana

Nous pourrions penser à l'intégration de l'outil de supervision Grafana qui permettra la supervision de nos différents objets. Un module de Django permettant d'intégrer Grafana facilement dans notre code a déjà été développé : il se trouve sur le github suivant : github.com/korfuri/django-prometheus.

4.2.3 Visualisation des métriques

L'interface Web pourra enfin permettre à l'utilisateur.rice de paramétrer et d'afficher de visualiser les métriques. Pour chaque métrique, on pourra sélectionner si on souhaite l'utiliser ou non (case à cocher) sur les stations/canaux qui nous intéressent. On pourra ensuite visualiser leurs valeurs graphiquement par station/canal.

4.2.4 Visualisation de l'état de l'archive finale

L'interface Web permettra également de visualiser l'état de l'archive finale. Une première page permettra d'avoir une vue globale de l'archive sous la forme d'un calendrier, avec différentes couleur en fonction de l'état de l'archive.

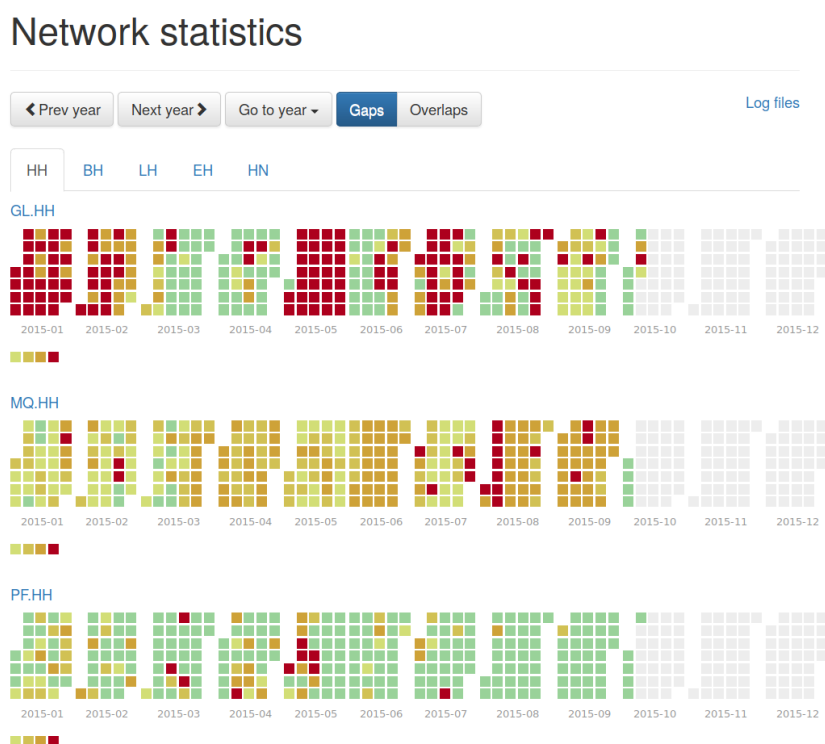


FIGURE 11 – Visualisation de l'archive finale – Source : ipgp.fr/satriano/stats/index.html?year=2015

Lorsque l'utilisateur.rice ira cliquer sur une journée, il aura alors le détail des stations et des canaux.

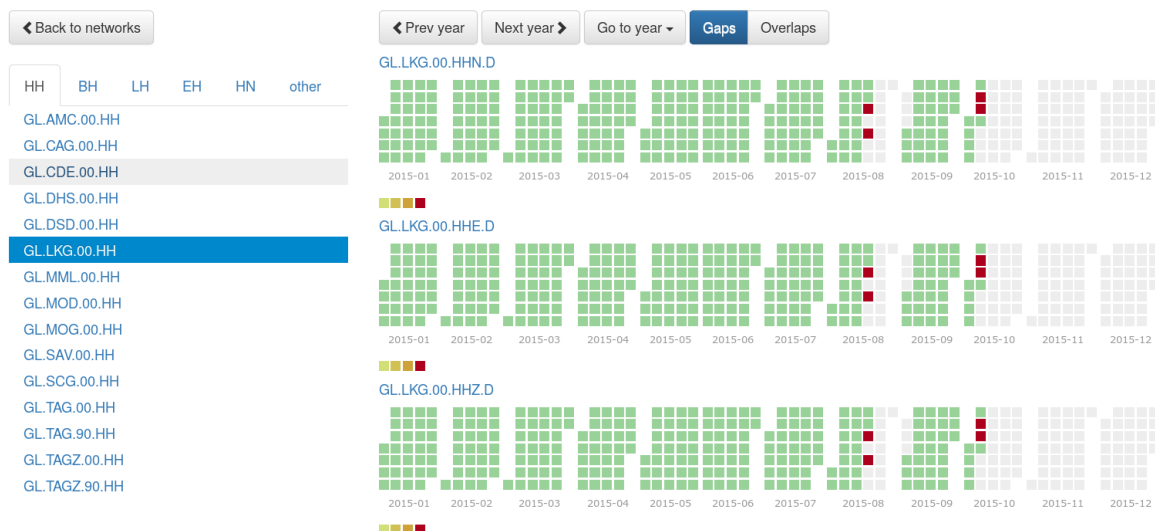


FIGURE 12 – Visualisation d'une station – Source : ipgp.fr/satriano/stats/stats/GL

Cette interface nous permettra de visualiser les différentes métriques. Pour chaque métrique, on associera une échelle de couleur afin d'avoir l'information visuellement et rapidement. Par exemple, pour les trous, on a :

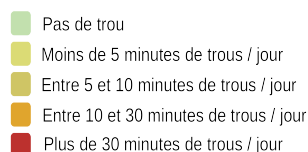


FIGURE 13 – Légende calendrier – Source : ipgp.fr/satriano/stats/index.html?year=2015

L'utilisateur.rice aura par ailleurs la possibilité de rendre silencieuse la surveillance d'une station. Pour plus de détails, voir 4.3.3.

4.2.5 Visualisation des tâches en cours

L'outil de visualisation permettra en outre d'accéder à la visualisation de l'état de la pile de gestion.

4.2.6 Visualisation des fichiers logs et des erreurs

La visualisation web permettra également d'avoir accès rapidement aux journal d'événements du logiciel afin d'analyser les erreurs et avertissements inhérents au fonctionnement du logiciel. Il sera également possible d'afficher le journal d'événements des traitements. Chacune de ces visualisation s'accompagnera d'une fonction de recherche afin de retrouver facilement des erreurs/avertissements. L'utilisateur.rice pourra en outre choisir d'afficher uniquement les erreurs ou les erreurs et avertissements.

4.3 Opérations manuelles

L'interface Web permettra d'ajouter une requête manuelle dans le cas où l'utilisateur.rice souhaiterait synchroniser des données depuis une source « offline » et forcer leur écriture dans l'archive finale, mais également de prendre la main sur les traitements en cours :

4.3.1 Récupération des données manuelles

Le logiciel SiQaCo permettra d'ajouter une nouvelle source manuellement depuis laquelle il ira charger des données. Cette nouvelle source pourra être soit temporaire, par exemple s'il s'agit d'une carte SD, soit permanente, par exemple s'il s'agit d'une nouvelle station. L'utilisateur.rice pourra également forcer la lecture de données ou forcer une requête à être effectué depuis les sources déjà définies, dans le cas où le besoin est urgent (crise sismique par exemple).

4.3.2 Annuler ou relancer une ou plusieurs requêtes

Dans la page de visualisation de la pile de requête, l'utilisateur.rice pourra choisir d'annuler ou de relancer une ou plusieurs requête(s).

Si l'utilisateur.rice choisi de relancer une requête (dont le statut est *on_hold*), son statut repasse à *retry* et sa durée de vie est réinitialisée.

4.3.3 Désactiver les alertes

Lors de l'analyse de la disponibilité des sources (cf. paragraphe 3.8.2), l'absence de disponibilité d'une station ou d'un canal lève une alerte (l'utilisateur.rice configure le nombre de canaux attendus après lecture d'une source afin de lever une alerte si il manque des données). L'utilisateur.rice pourra choisir de désactiver (en fait de les rendre silencieuses) ces alertes, par exemple si elles sont récurrentes pour une station/canal qu'il.elle sait ne pas fonctionner, ou si trop de requêtes échouent. Le logiciel devra cependant notifier à l'utilisateur.rice quelles sont les stations dont la surveillance a été désactivée (afin de ne pas "oublier" qu'elle ne fonctionne pas).

4.3.4 Modifications manuelles de l'archive finale

Le logiciel permettra également à l'utilisateur.rice d'intervenir sur les données de l'archive finale (cf. paragraphe 5.7 du cahier des charges) afin de :

- Supprimer un ou plusieurs fichier(s)
- Couper le début ou la fin d'un fichier afin de supprimer les données jugées non-valides
- Couper les données à l'intérieur d'un fichier (en gardant le début et la fin du fichier)

4.4 Choix de l'environnement de développement

L'utilisation du logiciel SiQaCo , sa configuration, les visualisations ainsi que la base de données seront toutes possibles via une interface web (locale et/ou distante).

Le choix de développer une interface web plutôt que d'une simple interface graphique s'est fait pour des raisons pratiques de maintenance. Il sera plus facile de faire évoluer cette interface qu'une interface graphique, et le logiciel sera plus simple à prendre en main pour de futurs développements.

Notre choix s'est porté sur l'environnement de développement Django. En effet, comme les traitements que nous effectuerons sont fondés sur la librairie Obspy, nous développerons le logiciel SiQaCo en langage Python. Il existe deux environnements de développement pour développer des interfaces web en Python : Flask et Django. La documentation de Django est plus fournie que celle de Flask, et il existe une multitude de paquets Django (disponibles sur djangopackages.org/). De plus, l'ORM (cf. 5.1) est intégré à Django, là où il faut faire appel explicitement à SQLAlchemy dans Flask (ce qui complique la tâche des développeurs présents et futurs).

L'environnement de travail Django utilise le vocabulaire suivant ¹ :

1. Ce vocabulaire est utilisé pour l'environnement de développement Django, il diffère de celui utilisé par PHP/Symphony qui utilise plutôt le schéma : MCV - modèle, contrôleur, vue, et qui sont respectivement dans Django les modèle, vue, gabarit

- Un **Modèle** comprendra la définition des classes avec leurs méthodes et attributs. C'est ce modèle qui sera interprété par l'ORM de Django afin de générer les tables correspondant à ces classes dans la base de données de SiQaCo .
- Une **Vue** est l'entité où s'effectueront les traitements : c'est ici que nous instancierons les objets, que nous appellerons leurs méthodes etc.
- Un **Gabarit** correspond à la visualisation. Les gabarits sont en général écrits en HTML.

Voici des exemples de vue, modèle et gabarit :

```

1 import datetime
2 from django.db import models
3 from django.utils import timezone
4
5 class Question(models.Model):
6     question_text = models.CharField(max_length=200)
7     pub_date = models.DateTimeField('date published')
8     def __str__(self):
9         return self.question_text
10    def was_published_recently(self):
11        # return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
12        now = timezone.now()
13        return now >= self.pub_date >= now - datetime.timedelta(days=1)
14    • was_published_recently.admin_order_field = 'pub_date'
15    • was_published_recently.boolean = True
16    • was_published_recently.short_description = 'Published recently?'
17
18 class Choice(models.Model):
19     question = models.ForeignKey(Question, on_delete=models.CASCADE)
20     choice_text = models.CharField(max_length=200)
21     votes = models.IntegerField(default=0)
22    def __str__(self):
23        return self.choice_text

```

FIGURE 14 – Exemple d'un modèle – Source : docs.djangoproject.com/fr/2.0/intro

```

80
81 def vote(request, question_id):
82     question = get_object_or_404(Question, pk=question_id)
83     try:
84         selected_choice = question.choice_set.get(pk=request.POST['choice'])
85     except (KeyError, Choice.DoesNotExist):
86         # Redisplay the question voting form.
87         return render(request, 'polls/detail.html', {
88             'question': question,
89             'error_message': "You didn't select a choice.",
90         })
91     else:
92         selected_choice.votes += 1
93         selected_choice.save()
94     # Always return an HttpResponseRedirect after successfully dealing
95     # with POST data. This prevents data from being posted twice if a
96     # user hits the Back button.
97     • return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))
98
99
100
101
102
103
104

```

FIGURE 15 – Exemple d'une vue – Source : *idem fig. 14*

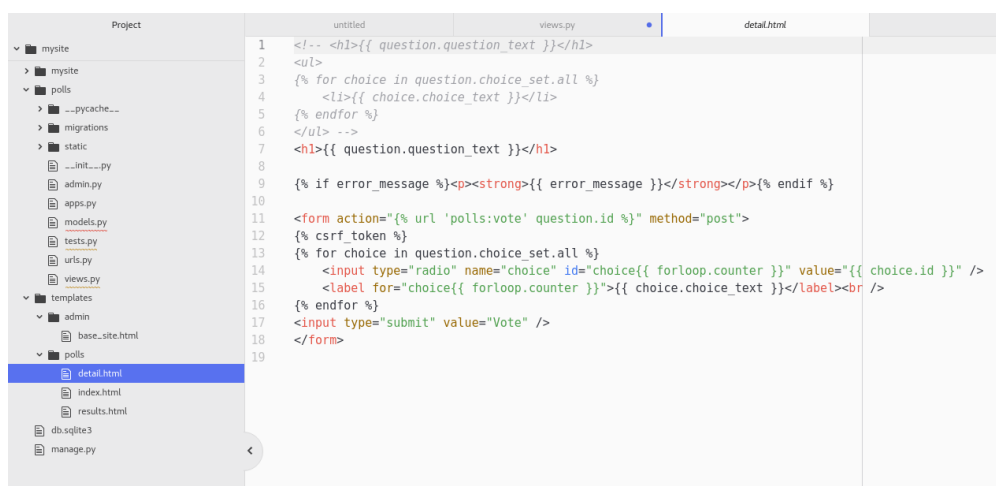


FIGURE 16 – Exemple d’un gabarit – *Source : idem fig. 14*

4.5 Lancement des scripts via un terminal

L’environnement de développement Django comprend une interface de programmation qui permettra à l’utilisateur.rice de lancer des scripts manuellement sans avoir à utiliser l’interface graphique. Afin que chacun puisse choisir s’il souhaite utiliser ou non l’interface graphique pour effectuer les traitements (traitement des discontinuités et contrôle qualité), l’interface graphique ne sera pensée que comme un outil visuel qui permettra de choisir les options avec lesquels seront effectués les traitements.

Le logiciel sera programmé afin d’obtenir différents niveaux de verbosité pour chaque commande, ainsi que des fichiers d’aide lorsque l’on appellera une commande de la façon suivante :

```
# command --help
```

Chapitre 5

Bases de données

Le logiciel SiQaCo intègre une base de données MySQL qui contiendra plusieurs tables et sera accessible via un compte “siqaco_admin” et un mot de passe défini par l'utilisateur.rice.

L'installation du logiciel nécessitera donc d'installer (si ce n'est déjà fait) le système de gestion de base de données relationnelle (SGBDR) MySQL. Ce choix à été fait vis à vis de la taille de la base de données (une base SQLite ne sera pas assez grande) et de l'utilisation qui en sera faite. Nous avons en effet besoin d'une base de données relationnelle, au vu de la diversité des données et des tables que nous allons avoir à définir, utiliser, enregistrer, et le choix de MySQL plutôt que PostgreSQL (par exemple) à été fait vis à vis de la facilité d'installation et de configuration de MySQL qui facilitera le déploiement de SiQaCo .

5.1 Outil de gestion de base de données

Afin de faciliter la gestion de la base de données de SiQaCo , nous utiliserons un ORM (Object Relational-Mapping) qui permettra de définir “des correspondances entre les schémas de la base de données et les classes du programme applicatif”¹. Notre choix s'est porté sur Django qui permet d'interfacer la gestion de cette base de données en Python, mais surtout car ce logiciel sera également utilisé pour le développement de notre interface Web (cf. Chapitre 4).

Ainsi, l'ensemble des tables que nous allons rappeler plus bas découle de l'instanciation des classes que nous avons définies plus haut dans ce document (cf. paragraphe 3.14).

Remarque : Il faut noter que chaque objet étant une entité d'une table de la base de données de SiQaCo , il lui sera automatiquement associé une clé d'identification unique.

5.2 Gestion des utilisateurs

L'utilisation de SiQaCo se fera par différents types d'utilisateurs. Nous allons donc créer différent types de droits, et lors de la mise en production, chaque utilisateur.rice rejoindra un des groupes suivants :

- Admin : cet.te utilisateur.rice pourra utiliser toutes les fonctionnalités de SiQaCo
- User : cet.te utilisateur.rice pourra utiliser les toutes les fonctionnalités de SiQaCo sauf la surcharge
- Viewer : cet.te utilisateur.rice pourra simplement utiliser SiQaCo afin de visualiser les traitements en cours, l'état de l'archive finale et pourra relancer le logiciel.

5.3 Tables

Voici la liste des différentes tables de la base de données de SiQaCo :

1. Source : [Wikipedia](#)

- Disponibilité des sources
- Statistique de la disponibilité des données sur une source (cette table permettra à l'utilisateur.rice de voir rapidement quel est le pourcentage de disponibilité des stations par jour/mois/année)
- Disponibilité des données sur une source
- État de l'archive finale (discontinuités)
- Pile des requêtes
- Historique des requêtes terminées (qu'elles aient réussi ou non)
- Disponibilité des ressources (CPU disponible, bande passante, etc. Définies a priori par l'utilisateur.rice mais on peut imaginer qu'elles soient mise à jour automatiquement par un module d'analyse des ressources lors d'un développement futur)
- Tables propres à l'ORM (cf. 5.1)
- Tables de gestion propres à l'utilisation de SiQaCo
- Table de la configuration du logiciel
- Table des métriques

Chapitre 6

Planning du développement

6.1 Développement

La première phase du développement, est la phase de conception, qui sera terminée avec la publication de ce dossier de conception. Elle comprends en particulier la conception des objets qui vont définir l'architecture du logiciel.

Dans un second temps, nous allons développer le logiciel SiQaCo .

La première étape du développement sera de définir les classes depuis lesquelles nos objets seront instanciés. Ensuite, nous définirons l'interface d'entrée/sortie des modules de lecture des sources.

Ensuite, nous allons développer le module de création des requêtes, puis la pile afin de commencer à peupler une archive dite "finale" locale.

Nous développerons ensuite les modules d'analyse de l'archive finale et de disponibilité et d'état des sources.

En parallèle, nous allons développer les premières briques de l'interface Web (locale pour l'instant) qui permettra les premières configurations telles que la configuration des sources, des fenêtre et granularité de l'analyse, ainsi que la visualisation de l'archive "finale".

Une fois ce premier développement terminé, nous aurons toutes les briques de bases servant au développement des futurs modules. La première étape du logiciel sera dès lors téléversée sur Gitlab.

Nous développerons alors les modules de contrôle de qualité de l'archive finale et d'analyse et visualisation de métriques sur l'archive finale.

Lorsque l'ensemble des modules permettant les traitements automatiques (pour le traitement des discontinuités et le contrôle qualité) seront développés, nous téléverserons le deuxième livrable sur Gitlab.

Puis nous aborderons la partie surcharge du logiciel, c'est à dire tous les modules qui permettrons à l'utilisateur.rice de prendre la main sur les traitements à savoir :

- Ajouter une nouvelle requête
- Annuler une requête
- Désactiver les alertes
- Modifier l'archive finale à la main (supprimer des bouts)

Une fois tous ces développements terminés et les tests terminés (cf. ci-dessous), nous passerons en phase de production.

6.2 Test

L'écriture de tests se fera tout au long du développement, afin de faciliter de futurs développements. Les tests seront écrits d'après la procédure décrite dans le tutoriel d'introduction à Django "Introduction aux tests automatisés" : docs.djangoproject.com/fr/2.0/intro/tutorial05/

Lorsque le développement de chaque livrable sera terminé, nous procéderons à une phase de test locale, et nous allons créer une archive finale a partir des données de Géoscope, et une autre à partir du Web Service de l'IRIS.

6.3 Production

Lorsque la phase de test sera terminée, le logiciel pourra être mis en ligne et installé et utilisé dans les différents observatoires.

6.4 Maintenance

La maintenance devra pouvoir se faire facilement, notamment grâce à :

1. Une documentation concise et exhaustive
2. Un code bien commenté
3. Les journaux d'événements du logiciel (relatif à son exécution) permettant de trouver la source d'erreur rapidement

Annexe A

Scripts existant de validation de données

A.1 Validation des métadonnées

- Vérification de la mise à jour quotidienne des dataless dans le répertoire SVN de volobsis (GetDataless.sh)
- Vérification de la consistance du dataless (Check_dataless.sh) qui utilise l'outil *verseed*
- Vérification de la location des stations avec Google Earth (PlotStaGoogleEarth.sh)
- Vérification de la réponse fréquentielle (PlotRESP.sh)
- Vérification de la consistance de l'arborescence SDS avec le dataless (Check_SDSdataless.sh)

A.2 Validation des données continues

1. Récupération des trous à partir du stockage des station
2. Liste des recouvrements (Check_SDS.sh)
3. Affichage de la complétude (PlotCompletenessData.sh)
4. Comparaison avec des séismes artificiels¹

1. Trois télé-séismes à une distance comprise entre 40° et 85° et de magnitude comprise entre 6.8 et 8.2 sont sélectionnés pour chaque année et pour chaque zone géographique (Antilles et Réunion). La méthode SCARDEC d'inversion du tenseur sismique est alors utilisée pour modéliser les formes d'onde sur les stations large-bande et moyenne-bande du réseau et les comparer aux données réelles déconvoluées. Pour les stations hébergeant un accéléromètre et un sismomètre, une comparaison de l'enregistrement déconvolué en vitesse des deux capteurs est effectuée pour des séismes régionaux significatifs

Annexe B

Références

B.1 Sites Web

| | |
|--|---|
| IRIS System for Portable Assessment of Quality | github.com/iris-edu/ispaq |
| EIDA WaveFormCatalog | orfeus-eu.org/data/eida/webservices/wfcatalog/ |
| The Mini-SEED library | github.com/iris-edu/libmseed |
| Grafana, outil de monitoring et de visualisation de données | grafana.com |
| Obspy | obspy.org |
| Anaconda | anaconda.com |
| Django | djangoproject.com |
| Ubuntu | ubuntu.com |
| Atom | atom.io |
| git | git-scm.com |

Annexe C

Glossaire

C.1 Noms

| | |
|--------------|--|
| RESIF | Réseau Sismologique et Géodésique Français |
| IPGP | Institut de Physique du Globe de Paris |
| EOST | École et Observatoire des Sciences de la Terre (Strasbourg) |
| OVPF | Observatoire Volcanique du Piton de la Fournaise (Île de la Réunion) |
| OVSG | Observatoire Volcanique et Sismologique de la Guadeloupe |
| OVSF | Observatoire Volcanique du Piton de la Martinique |
| AEQC | Outil de validation des données |
| SEED | Standart for the Exchange of Earthquake Data |
| SDS | Seiscomp Data Structure |
| FDSN | International Federation of Digital Seismograph Networks |

C.2 Abréviations

| | |
|--------------|---|
| SGBDR | Système de Gestion de Base de Données Relationnel |
| ORM | <i>Object Relationnal Mapper</i> |
| IDE | <i>Integrated Development Environment</i> |
| FIFO | <i>First In, First Out</i> |
| LIFO | <i>Last In, First Out</i> |