Cahier des Charges - Application Synchro Qt Made

1. Introduction

Ce document décrit les fonctionnalités et les exigences techniques de l'application "Synchro Qt Made", un outil de sauvegarde et de synchronisation de répertoires. Il intègre toutes les spécifications initiales ainsi que les améliorations et corrections apportées au cours du développement.

2. Objectif Général

L'objectif principal de l'application est de permettre aux utilisateurs de configurer, lancer, surveiller et gérer des tâches de synchronisation de fichiers et de répertoires entre une source locale et une destination sur un volume externe, avec des options de fréquence, de filtrage et de gestion des versions.

3. Architecture de l'Application

L'application est conçue avec une architecture client-serveur simplifiée, composée de trois composants principaux :

- **Frontend (Interface Utilisateur - mainwindow.py) :** Une application de bureau basée sur PyQt5, responsable de l'interaction utilisateur, de la collecte des configurations et de l'affichage des statuts et logs.
- **Backend (API REST - api.py) :** Un serveur Flask léger qui expose une API REST pour gérer les configurations et contrôler les processus de synchronisation. Il sert d'intermédiaire entre le frontend et le moteur de synchronisation.
- **Moteur de Synchronisation (sync_engine.py) :** Un script Python exécuté en tant que processus séparé par le backend, responsable de la logique réelle de copie et de synchronisation des fichiers, de la gestion de la fréquence et du cache des versions.

4. Fonctionnalités Détaillées

4.1. Gestion des Configurations

- **Création et Sauvegarde :**
  - L'utilisateur peut définir une configuration de synchronisation via l'interface.
  - Les configurations sont sauvegardées au format JSON dans le répertoire ~/.synchro/configs/.
  - Le nom de la configuration est dérivé du nom du répertoire source (ex: si source est /home/user/Documents/ProjetX, le nom est ProjetX).
- **Chargement :**

    - L'utilisateur peut charger une configuration existante via un dialogue de sélection de fichier.
    - **Chargement au Démarrage :** Au démarrage de l'application, la configuration nommée "BacASable.json" est automatiquement chargée si elle existe.
- **Suppression :**
    - L'utilisateur peut supprimer une configuration existante après confirmation.
- **Champs de Configuration :**
    - **Source :** Chemin du répertoire à synchroniser (saisie manuelle ou via bouton "Parcourir").
    - **Destination :** Chemin du répertoire de destination. Ce chemin est généré automatiquement en combinant un chemin de base configurable (external_volume_base_path dans volume_config.json) et le nom du répertoire source. Le champ est non éditable.
    - **Fréquence (heures) :** Intervalle en heures entre chaque synchronisation (saisie numérique et slider, de 1 à 24 heures).
    - **Blacklist Fichiers :** Liste de motifs de fichiers à exclure de la synchronisation (séparés par des points-virgules).
    - **Blacklist Répertoires :** Liste de motifs de répertoires à exclure de la synchronisation (séparés par des points-virgules).
    - **Max Versions en Cache : NOUVEAU** Nombre maximal de versions antérieures d'un fichier modifié à conserver dans le répertoire de destination (saisie numérique de 0 à 99). Cette valeur est chargée depuis volume_config.json et transmise au backend/moteur de synchronisation.

4.2. Gestion des Synchronisations

- **Démarrage :**
    - L'utilisateur lance une synchronisation pour la configuration actuellement affichée.
    - Le frontend envoie la configuration complète au backend.
    - Le backend lance le sync_engine.py en tant que processus séparé, lui passant tous les paramètres de la configuration.
    - Le backend gère le PID du processus de synchronisation.
- **Arrêt :**
    - L'utilisateur peut envoyer un signal d'arrêt à la tâche de synchronisation en cours pour la configuration affichée.
    - Le backend tente d'arrêter le processus de manière propre (SIGTERM), puis de le tuer si nécessaire.
- **Fréquence :**
    - Le moteur de synchronisation exécute la tâche, puis attend la durée spécifiée par la fréquence avant de relancer une nouvelle passe.
- **Blacklists :**
    - Le moteur de synchronisation exclut les fichiers et répertoires correspondant aux motifs définis dans les blacklists.

- **Gestion du Cache des Versions : NOUVEAU**
    - Le moteur de synchronisation doit implémenter une logique pour conserver uniquement le nombre spécifié de versions antérieures (max_cached_versions) des fichiers modifiés dans le répertoire de destination, en supprimant les plus anciennes si le nombre est dépassé.

4.3. Interface Utilisateur (Frontend - mainwindow.py)

- **Structure Générale :** Fenêtre principale (QMainWindow) avec un agencement clair des éléments (QVBoxLayout, QHBoxLayout, QFormLayout).
- **Éléments Visuels :**
    - Titre principal ("Sauvegarde / Synchronisation").
    - Label affichant le chemin de destination calculé (#destinationPathLabel).
    - Champs de saisie pour source, fréquence, blacklists, max versions cache.
    - Slider pour la fréquence.
    - Ligne de séparation graphique.
    - Barre de progression.
    - Zone de log.
- **Boutons d'Action :**
    - "Charger Configuration", "Sauvegarder Configuration", "Détruire Configuration".
    - "Start" (vert par défaut, devient rouge et désactivé si synchro en cours).
    - "Stop" (désactivé par défaut, devient actif si synchro en cours).
    - "Synthèse" (affiche un résumé des tâches de synchro).
    - "Clear Log", "Copie Log".
    - "Quitter".
- **Zone de Log :**
    - QTextBrowser en lecture seule.
    - Limité à 100 lignes pour éviter la surcharge mémoire.
    - Supporte les liens cliquables (pour ouvrir les fichiers de log).
    - Messages horodatés, avec indication INFO/ERREUR et codes d'erreur.
- **Barre de Progression (QProgressBar) :**
    - Visible uniquement pendant une synchronisation active pour la configuration affichée.
    - Affiche le pourcentage de progression.
    - À la fin de la synchronisation (terminée, arrêtée, erreur), affiche un message de statut clair ("Synchronisation terminée", "arrêtée", "Erreur").
    - Reste visible 3 secondes après la fin de la tâche, puis se masque.
- **Mise à Jour de l'UI en Temps Réel :**
    - Un QTimer interroge le backend toutes les 2 secondes pour obtenir le statut des tâches de synchronisation.
    - **Mise à jour spécifique à la configuration sélectionnée :** Lorsque l'utilisateur charge une configuration, le frontend interroge le backend pour l'état de la synchronisation *correspondant à cette configuration*.
        - Si la synchronisation de la configuration sélectionnée est running, l'UI

(bouton "Start" rouge, barre de progression active) est mise à jour en conséquence.
        - Si la synchronisation de la configuration sélectionnée n'est *pas* running (soit elle est terminée, arrêtée, en erreur, soit elle n'a jamais été lancée), l'UI est réinitialisée à l'état "idle" (bouton "Start" vert, barre de progression masquée) et un message est affiché dans le log.
        - Si une tâche active pour la configuration sélectionnée passe à un état final (completed, stopped, error), l'UI est mise à jour et la synthèse finale est affichée.
    - Le timer est démarré si au moins une tâche est active sur le backend et arrêté si aucune tâche n'est active.
- **Styles :** L'apparence de l'interface est configurable via gui_config.json (couleurs, polices, tailles, styles des boutons).

4.4. Backend (API REST - api.py)

- **Framework :** Flask avec Flask-CORS pour les requêtes cross-origin.
- **Endpoints API :**
    - PUT /api/configs/<config_name> : Sauvegarde ou met à jour une configuration.
    - POST /api/sync_tasks/start/<config_name> : Démarre une tâche de synchronisation.
    - POST /api/sync_tasks/stop/<config_name> : Arrête une tâche de synchronisation.
    - GET /api/sync_tasks : Récupère le statut de toutes les tâches de synchronisation actives et récemment terminées, incluant la progression, la durée et les statistiques détaillées (répertoires/fichiers ajoutés/modifiés/supprimés, nom du fichier de log).
- **Gestion des Tâches :** Utilise une classe TaskManager pour suivre les objets SyncTask (une par configuration lancée).
- **Lancement des Processus :** Utilise subprocess.Popen pour lancer sync_engine.py en arrière-plan, redirigeant sa sortie vers un fichier de log spécifique à la tâche.
- **Parsing des Logs de Tâches :** La classe SyncTask analyse son propre fichier de log pour extraire la progression et les statistiques de synthèse (répertoires/fichiers ajoutés/modifiés/supprimés) à l'aide d'expressions régulières robustes.
- **Nettoyage du Log :** Les messages de débogage des requêtes API sont journalisés au niveau DEBUG pour ne pas polluer les logs en production.
- **Gestion des PIDs :** Les PIDs des processus de synchronisation sont capturés et exposés via l'API.

4.5. Moteur de Synchronisation (sync_engine.py)

- **Logique de Synchronisation :** Doit implémenter la logique de copie/mise à jour des fichiers de la source vers la destination, en tenant compte des blacklists.
- **Journalisation :** Écrit des messages de progression et de synthèse détaillés dans son fichier de log spécifique à la tâche.
- **Gestion de la Fréquence :** Après chaque cycle de synchronisation, le script doit

attendre la durée spécifiée par le paramètre frequency_hours avant de relancer une nouvelle passe.
- **Gestion du Cache des Versions : NOUVEAU** Lors de la modification d'un fichier existant, le moteur doit sauvegarder les versions antérieures dans un sous-répertoire de cache (ex: .versions/) et supprimer les versions les plus anciennes si le nombre dépasse max_cached_versions.

5. Fichiers de Configuration et de Données

- ~/.synchro/ : Répertoire principal de l'application (créé automatiquement).
  - configs/ : Contient les fichiers JSON des configurations de synchronisation.
  - logs/ : Contient les fichiers de log.
    - app.log : Log du frontend.
    - backend_app.log : Log du backend.
    - tasks/ : Contient les fichiers de log spécifiques à chaque tâche de synchronisation (nommés [config_name]_[timestamp].log).
  - volume_config.json : Fichier JSON pour les configurations globales du volume externe.
    - external_volume_base_path : Chemin de base pour les destinations de synchronisation.
    - max_cached_versions : **NOUVEAU** Nombre maximal de versions de fichiers à conserver dans le cache.
- gui_config.json : Fichier JSON situé à côté de mainwindow.py, définissant les styles CSS de l'interface utilisateur.

6. Exigences Non Fonctionnelles

- **Robustesse :**
  - Gestion des chemins invalides, permissions, fichiers verrouillés.
  - Capacité à récupérer l'état des synchronisations après un redémarrage de l'interface ou du backend.
  - Gestion des interruptions (arrêt brutal de l'application, déconnexion réseau).
- **Performance :**
  - Gestion efficace des grands volumes de données et des nombreux fichiers.
  - Comportement stable avec des fréquences de synchronisation élevées.
- **Expérience Utilisateur (UI/UX) :**
  - Interface claire, intuitive et esthétique (thème sombre, espacement, typographie).
  - Utilisation d'icônes pour améliorer la compréhension visuelle.
  - Animations subtiles pour les changements d'état (ex: barre de progression).
  - Notifications système pour les événements importants (fin de synchro).
  - Filtrage et coloration des messages dans la zone de log.
- **Gestion des Logs : NOUVEAU**
  - Les fichiers de log situés dans ~/.synchro/logs/ (incluant app.log, backend_app.log et ceux dans tasks/) ne doivent pas excéder une durée de vie

d'une semaine (7 jours).
    - Le nettoyage des logs du backend (backend_app.log et tasks/) est géré automatiquement par le backend.
    - Le nettoyage du log du frontend (app.log) est géré automatiquement par le frontend.
- **Gestion de Version :** Le projet doit être géré via Git et hébergé sur GitHub, avec un fichier .gitignore approprié pour exclure les fichiers non essentiels (environnements virtuels, logs, configurations locales).

7. Outils et Technologies Utilisés

- **Langage de Programmation :** Python (version 3.x)
- **Frontend :** PyQt5
- **Backend (API REST) :** Flask
- **Extensions Backend :** Flask-CORS
- **Gestion de Processus :** Module subprocess de Python
- **Manipulation de Chemins :** Module pathlib de Python
- **Format de Données :** JSON
- **Contrôle de Version :** Git
- **Hébergement de Dépôt :** GitHub