

EVA informatique 2021 : Chaos Monkey, Linux et Python

Proposition de corrigé — non officielle. Session 2021, sujet 2 : les trois parties, dans l'option Unix, sous Linux avec Python 3.

[Sujet officiel](#) · [Annales de ce concours ou examen](#)

Hypothèses retenues

Système : Unix, sous Linux. Langage : Python 3. Les exemples utilisent `psutil` pour les processus et les volumes, et `requests` pour le client HTTP. Les fonctions perturbatrices supposent un environnement de test isolé et les privilèges nécessaires ; elles ne sont pas lancées par ce corrigé. Une erreur d'accès ou une mesure absente produit une erreur explicite.

La couverture annonce quatre heures et un coefficient 4, alors que les pieds de page indiquent deux heures et un coefficient 2. Le document comporte cette incohérence ; les trois parties sont néanmoins traitées.

Partie 1 — Chaos Monkey et Doctor Monkey

1. Arrêter brutalement quatre processus

`kill()` envoie `SIGKILL` sous Unix, contrairement à un arrêt gracieux par `SIGTERM`. On choisit quatre PID distincts parmi les processus autorisés pour le test, en excluant PID 1 et le programme lui-même. L'appelant fournit la liste autorisée ; il faut au moins quatre processus encore présents. Une disparition entre sélection et arrêt est possible et reste signalée.

```
import os
import random
import subprocess
import time
from pathlib import Path
import psutil

def arreter_quatre(pids_autorises):
    possibles = sorted(set(pids_autorises) & set(psutil.pids()))
    possibles = [p for p in possibles if p not in (1, os.getpid())]
    if len(possibles) < 4:
        raise ValueError("Moins de quatre processus admissibles")
    choisis = random.sample(possibles, 4)
    erreurs = {}
    for pid in choisis:
        try:
            psutil.Process(pid).kill()
        except (psutil.NoSuchProcess, psutil.AccessDenied) as e:
            erreurs[pid] = str(e)
    return {"ok": not erreurs, "choisis": choisis, "erreurs": erreurs}
```

2. Désactiver uniquement les interfaces Ethernet

Sous Linux, le type ARPHRD_ETHER vaut 1, mais des interfaces Wi-Fi emploient aussi ce type. On exclut donc celles qui exposent un répertoire wireless, ainsi que le bouclage. Le périmètre retenu inclut les interfaces Ethernet virtuelles ; si le test vise uniquement des cartes physiques, il faut également vérifier leur rattachement à un périphérique matériel.

```
def couper_ethernet():
    interfaces = []
    for p in Path("/sys/class/net").iterdir():
        if p.name == "lo" or (p / "wireless").exists():
            continue
        if (p / "type").read_text().strip() == "1":
            subprocess.run(["ip", "link", "set", "dev", p.name,
                           "down"], check=True)
            interfaces.append(p.name)
    return {"ok": True, "interfaces": interfaces}
```

3. Démonter le plus grand disque

Un disque physique peut porter plusieurs partitions et points de montage. Ici, « disque » est interprété comme **volume de stockage monté** : on compare la capacité totale des systèmes de fichiers, pas l'espace encore libre. Si plusieurs montages ont la même capacité, le tri fixe un choix reproductible. Le démontage normal peut être refusé si le volume est occupé ; on ne force pas silencieusement la destruction d'opérations en cours.

```
def demonter_plus_grand_volume():
    volumes = []
    for p in psutil.disk_partitions(all=False):
        taille = psutil.disk_usage(p.mountpoint).total
        volumes.append((taille, p.mountpoint))
    if not volumes:
        raise ValueError("Aucun volume monte")
    _, montage = max(volumes)
    subprocess.run(["umount", "--", montage], check=True)
    return {"ok": True, "montage": montage}
```

Pour viser le **disque physique** le plus volumineux, on utiliserait l'inventaire lsblk pour comparer les équipements de type disk, puis démonter chacun de leurs systèmes de fichiers. Cette variante requiert un traitement des volumes logiques et des montages multiples ; elle ne se confond pas avec la fonction ci-dessus.

4. Arrêter un serveur démarré depuis plus de deux jours

Deux jours valent 172 800 secondes. La comparaison est strictement > : exactement deux jours ne déclenche pas l'arrêt. /proc/uptime fournit le temps depuis le démarrage.

```
def arreter_si_ancien():
    uptime = float(Path("/proc/uptime").read_text().split()[0])
    if uptime > 2 * 24 * 3600:
        subprocess.run(["shutdown", "-h", "now"], check=True)
        return {"ok": True, "arret_demande": True}
    return {"ok": True, "arret_demande": False}
```

5. Degrader les entrées-sorties sans couper le service

Pour les disques, on peut limiter le débit et les opérations par seconde d'un groupe de processus avec les cgroups, ou réduire leur priorité d'entrées-sorties avec `ionice` lorsque l'ordonnanceur utilisé la prend en compte. Une priorité plus faible ne garantit pas un débit fixe. Pour le réseau, `tc` permet de limiter la bande passante ou d'introduire un délai et une perte contrôlée avec `netem`.

Les paramètres restent bornés : on commence par un ralentissement modéré, on observe erreurs et latence, puis on retire la perturbation au franchissement d'un seuil convenu. Ni bloquer tous les paquets ni saturer totalement le disque ne répondrait à « sans interrompre le service ».

6. Vérifier une mémoire disponible d'au moins 5 %

`MemAvailable` estime la mémoire utilisable sans pagination, contrairement à `MemFree`, qui ne compte que les pages libres. Le seuil demandé inclut l'égalité.

```
def memoire_suffisante():
    valeurs = {}
    for ligne in Path("/proc/meminfo").read_text().splitlines():
        nom, valeur = ligne.split(":", 1)
        valeurs[nom] = int(valeur.strip().split()[0])
    total = valeurs["MemTotal"]
    if total <= 0:
        raise ValueError("MemTotal invalide")
    return valeurs["MemAvailable"] >= 0.05 * total
```

7. Dépassement de la capacité de traitement

La **charge moyenne**, disponible dans `/proc/loadavg`, constitue un indicateur existant. Une charge durablement supérieure au nombre de processeurs logiques peut signaler une file de tâches importante. Sous Linux, elle comprend aussi des tâches en attente non interruptible, notamment d'entrées-sorties : on la confronte à l'utilisation CPU et à l'attente disque, plutôt que d'affirmer qu'elle prouve à elle seule une saturation CPU.

8. Nombre de connexions sur un port

La commande suivante compte les connexions TCP **établies** dont le port local est 80 :

```
ss -Htan state established '( sport = :80 )'
```

La fonction compte les lignes, sans en-tête. Elle ne compte ni la socket d'écoute ni `TIME_WAIT` ; l'UDP n'a pas le même état de connexion.

```
def connexions_tcp(port):
    if not isinstance(port, int) or not 1 <= port <= 65535:
        raise ValueError("Port invalide")
    texte = subprocess.check_output(
        ["ss", "-Htan", "state", "established",
         "( sport = :%d )" % port], text=True)
    return len(texte.splitlines())
```

9. Déterminer qu'un serveur n'est pas utilisé

On observe une fenêtre de temps couvrant ses cycles d'activité : CPU, entrées-sorties, trafic, sessions, requêtes applicatives et travaux planifiés. Une activité basse à un instant ne signifie pas qu'un serveur est inutile : il peut assurer une astreinte, une réplication ou une sauvegarde nocturne. L'indicateur combine donc seuils techniques et absence de travail métier attendu, avec un état « inconnu » si la collecte manque.

Partie 2 — Sélection, parallélisme et rapports

1. Programme complet avec trois traitements simultanés

Il faut huit serveurs distincts et au moins quatre routines distinctes pour respecter la limite de deux utilisations. Dupliquer chaque nom de routine deux fois puis tirer huit éléments garantit cette limite. La fonction `lanceRoutine` est fournie par le sujet et gère déjà l'erreur ou le dépassement de trois minutes. Un pool de **trois processus** exécute les appels synchrones ; les résultats reviennent au processus principal, seul à écrire le journal.

```
import csv
from datetime import datetime, timezone
from concurrent.futures import ProcessPoolExecutor, as_completed

def executer(serveur, routine, lanceRoutine):
    debut = time.time()
    chrono = time.monotonic()
    try:
        code = int(lanceRoutine(serveur, routine) == 1)
    except Exception:
        code = 0
    duree = round(1000 * (time.monotonic() - chrono))
    return serveur, routine, code, debut, duree

def campagne(serveurs, routines, lanceRoutine):
    serveurs = list(dict.fromkeys(serveurs))
    routines = list(dict.fromkeys(routines))
    if len(serveurs) < 8 or len(routines) < 4:
        raise ValueError("Huit serveurs et quatre routines requis")
    cibles = random.sample(serveurs, 8)
    actions = random.sample(routines * 2, 8)
    fichier = datetime.now(timezone.utc).strftime("%Y%m%d.log")
    with open(fichier, "a", newline="", encoding="utf-8") as f:
        writer = csv.writer(f, delimiter="\t")
        with ProcessPoolExecutor(max_workers=3) as pool:
            futurs = [pool.submit(executer, s, r, lanceRoutine)
                      for s, r in zip(cibles, actions)]
            for futur in as_completed(futurs):
                writer.writerow(futur.result())
            f.flush()
    return fichier

if __name__ == "__main__":
    # Les listes et la fonction sont fournies par l'application.
    campagne(serveurs, routines, lanceRoutine)
```

Les fonctions `executer` et `lanceRoutine` doivent être définies au niveau du module, et non comme fonctions locales ou lambdas, pour être transmissibles aux processus. Le garde `__main__` empêche de

relancer le programme à l'import. Le début est un timestamp Unix ; la durée utilise une horloge monotone. Le fichier suit ici la date UTC du lancement. Une liste insuffisante est refusée avant toute action.

2. Exécution après une modification

Une chaîne d'intégration continue déclenche les tests après un changement du code ou du plan de perturbations. Elle vérifie d'abord la configuration, puis exécute une campagne sur un environnement identifié. Un déclenchement planifié complète celui du dépôt. Chaque campagne conserve version, paramètres, cibles et résultats afin de comparer deux exécutions.

3. Tableau de bord

Le tableau juxtapose la perturbation injectée et ses effets : chronologie, retours des routines, latences applicatives, taux d'erreurs, débit, disponibilité, ressources et durée de retour à l'état normal. Une commande réussie signifie seulement que la perturbation a été exécutée ; elle ne prouve pas la résilience. On compare aux mesures avant le test et à des objectifs de service.

4. Confidentialité chez l'hébergeur

Le **chiffrement homomorphe** permet certains calculs sur des données chiffrées : le résultat est ensuite déchiffré par le détenteur de la clé. Une somme de compteurs peut, par exemple, être obtenue sans révéler chaque compteur au prestataire. Le schéma doit supporter les opérations réellement nécessaires et ses coûts doivent être compatibles avec le volume. TLS protège le transport, et le chiffrement au repos le stockage ; ils ne permettent pas à eux seuls de calculer sur les données sans les déchiffrer. Une pseudonymisation peut conserver certains regroupements, sans offrir la même confidentialité.

Partie 3 — Programme principal et agents

1. Échanges réseau proposés

On retient HTTPS avec authentification mutuelle par certificats. Chaque demande contient un **identifiant unique** et un nom de routine appartenant à une liste fermée. L'agent accuse réception et lance l'action ; le principal consulte ensuite son résultat. Il ne transmet pas de commande shell arbitraire.

Voici les parties réseau d'un agent Python. ROUTINES est le dictionnaire des fonctions perturbatrices précédentes, configuré au démarrage. La table SQLite conserve l'identité d'une demande et interdit sa réexécution automatique. Si un agent redémarre avec un travail encore marqué en cours, son sort doit être réconcilié ; il n'est pas relancé à l'aveugle.

```
import json
import sqlite3
import ssl
import threading
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

DB = "agent-jobs.sqlite"

def stocker_resultat(job, routine):
    try:
```

```

        resultat = ROUTINES[routine]()
        code = int(isinstance(resultat, dict)
                    and resultat.get("ok") is True)
except Exception:
    code = 0
with sqlite3.connect(DB) as db:
    db.execute("UPDATE jobs SET etat='done', code=? WHERE id=?",
               (code, job))

class Agent(BaseHTTPRequestHandler):
    def repondre(self, status, objet):
        texte = json.dumps(objet).encode()
        self.send_response(status)
        self.send_header("Content-Type", "application/json")
        self.send_header("Content-Length", str(len(texte)))
        self.end_headers()
        self.wfile.write(texte)

    def do_POST(self):
        if self.path != "/run":
            return self.repondre(404, {"error": "route"})
        try:
            taille = int(self.headers.get("Content-Length", "0"))
            if not 0 < taille <= 8192:
                raise ValueError("taille")
            v = json.loads(self.rfile.read(taille))
            job, routine = v["job"], v["routine"]
            if not isinstance(job, str) or not 1 <= len(job) <= 80:
                raise ValueError("identite")
            if routine not in ROUTINES:
                raise ValueError("routine")
        except (ValueError, KeyError, TypeError):
            return self.repondre(400, {"error": "demande invalide"})
        with sqlite3.connect(DB) as db:
            try:
                db.execute("INSERT INTO jobs VALUES (?, ?, 'running', NULL)",
                           (job, routine))
                nouveau = True
            except sqlite3.IntegrityError:
                ancien = db.execute("SELECT routine FROM jobs WHERE id=?",
                                     (job,)).fetchone()[0]
                if ancien != routine:
                    return self.repondre(409, {"error": "identite reutilisee"})
                nouveau = False
        if nouveau:
            threading.Thread(target=stocker_resultat,
                            args=(job, routine), daemon=True).start()
        self.repondre(202, {"job": job})

    def do_GET(self):
        if not self.path.startswith("/jobs/"):
            return self.repondre(404, {"error": "route"})
        job = self.path[len("/jobs/"): ]
        with sqlite3.connect(DB) as db:
            ligne = db.execute("SELECT etat, code FROM jobs WHERE id=?",
                               (job,)).fetchone()
        if ligne is None:
            return self.repondre(404, {"error": "inconnu"})
        self.repondre(200, {"state": ligne[0], "code": ligne[1]})

```

```

def servir(adresse, cert, cle, ca):
    with sqlite3.connect(DB) as db:
        db.execute("CREATE TABLE IF NOT EXISTS jobs (
            'id TEXT PRIMARY KEY, routine TEXT, etat TEXT, code INTEGER)")
    tls = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    tls.load_cert_chain(cert, cle)
    tls.load_verify_locations(ca)
    tls.verify_mode = ssl.CERT_REQUIRED
    serveur = ThreadingHTTPServer(adresse, Agent)
    serveur.socket = tls.wrap_socket(serveur.socket, server_side=True)
    serveur.serve_forever()

```

Les routines perturbatrices partagent un contrat : un dictionnaire contenant ok, booléen. La routine d'arrêt de processus renvoie ok=False si au moins un arrêt échoue ; les erreurs ne deviennent donc pas un succès. Pour l'arrêt conditionnel du serveur, une condition non remplie est une vérification réussie sans arrêt. Les autres routines ajoutées au dictionnaire doivent respecter le même contrat ; une exception ou un retour incompatible produit le code 0.

L'autorité de certification est dédiée aux agents et orchestrateurs autorisés. Le filtrage réseau restreint les accès. Le principal conserve son suivi dans SQLite **avant tout POST**. Il réserve au plus trois instructions à la fois ; si leur fin reste incertaine, aucune nouvelle instruction n'est envoyée. Les agents continuent éventuellement leur action après le délai du client : on ne libère donc pas leur place simplement parce qu'une attente a expiré.

```

import uuid
import requests
from concurrent.futures import ThreadPoolExecutor

SUIVI = "principal-jobs.sqlite"

def preparer_suivi():
    with sqlite3.connect(SUIVI) as db:
        db.execute("CREATE TABLE IF NOT EXISTS jobs (
            'id TEXT PRIMARY KEY, url TEXT, routine TEXT, "
            "etat TEXT, code INTEGER)")

def noter(job, etat, code=None):
    with sqlite3.connect(SUIVI) as db:
        db.execute("UPDATE jobs SET etat=?, code=? WHERE id=?",
            (etat, code, job))

def lire_resultat(session, url, job):
    r = session.get(url + "/jobs/" + job, timeout=(3, 5))
    r.raise_for_status()
    v = r.json()
    if v.get("state") == "done" and v.get("code") in (0, 1):
        return int(v["code"])
    return None

def lancer_distant(url, routine, job, certificat, ca):
    session = requests.Session()
    session.cert, session.verify = certificat, ca
    fin = time.monotonic() + 180
    try:
        r = session.post(url + "/run",
            json={"job": job, "routine": routine},
            timeout=(3, 5))

```

```

        r.raise_for_status()
    except requests.RequestException:
        pass # Le même identifiant reste suivi ; pas de nouveau POST.
    while time.monotonic() < fin:
        try:
            code = lire_resultat(session, url, job)
            if code is not None:
                noter(job, "done", code)
                return {"job": job, "state": "done", "code": code}
            except (requests.RequestException, ValueError, AttributeError):
                pass
            time.sleep(1)
        noter(job, "uncertain")
    return {"job": job, "state": "uncertain", "code": None}

def principal(plan, certificat, ca):
    # plan : liste de couples (URL d'agent autorisée, nom de routine).
    preparer_suivi()
    plan = list(plan)
    resultats = []
    for i in range(0, len(plan), 3):
        lot = [(uuid.uuid4().hex, url, routine)
                for url, routine in plan[i:i+3]]
        with sqlite3.connect(SUIVI) as db:
            db.execute("BEGIN IMMEDIATE")
            actif = db.execute("SELECT 1 FROM jobs WHERE "
                               "etat IN ('pending', 'uncertain') "
                               "LIMIT 1").fetchone()

            if actif:
                raise RuntimeError("Instruction existante a reconcilier")
            db.executemany("INSERT INTO jobs VALUES "
                           "(?, ?, ?, 'pending', NULL)", lot)
        # Le commit du suivi précède tout envoi.
        with ThreadPoolExecutor(max_workers=3) as pool:
            futurs = [pool.submit(lancer_distant, url, routine, job,
                                   certificat, ca)
                      for job, url, routine in lot]
            for f in futurs:
                resultats.append(f.result())
        if any(r["state"] != "done" for r in resultats):
            return {"results": resultats, "remaining": plan[i+3:],
                    "blocked": True}
    return {"results": resultats, "remaining": [], "blocked": False}

def reconcilier(job, certificat, ca):
    with sqlite3.connect(SUIVI) as db:
        ligne = db.execute("SELECT url, etat FROM jobs WHERE id=?",
                           (job,)).fetchone()

    if ligne is None:
        raise ValueError("Identifiant inconnu")
    session = requests.Session()
    session.cert, session.verify = certificat, ca
    code = lire_resultat(session, ligne[0], job)
    if code is not None:
        noter(job, "done", code)
    return {"job": job, "code": code}

```

Les identifiants et URL restent consultables dans `principal-jobs.sqlite`, même après redémarrage. `reconcilier` interroge le même agent et le même travail, sans envoyer d'ordre. Un GET inconnu ou une

erreur réseau ne prouve pas l'arrêt de l'action : le suivi reste bloquant. Une action encore marquée en cours sur un agent redémarré demande également une vérification de son sort ; on ne la remplace pas automatiquement.

Le suivi SQLite est commun aux appels de ce principal. La transaction réserve le lot avant l'envoi et empêche deux appels concurrents d'ouvrir chacun trois places. Plusieurs principaux indépendants demanderaient un coordonnateur partagé. Ici, les threads de la partie réseau attendent les réponses HTTP ; **les processus exigés dans la partie 2 sont assurés par son pool de processus distinct**. La fin d'une attente réseau ne remplace pas la règle de délai appliquée par `lanceRoutine` dans cette partie.

2. Sécuriser le dispositif

La conception limite les cibles, les routines, les privilèges et la durée. Une identité authentifiée ne doit pas recevoir automatiquement tous les droits ; la configuration associe les certificats aux rôles autorisés. Les clés sont protégées, les actions journalisées, et un arrêt d'urgence permet de suspendre la campagne. On teste aussi la perte de l'agent, du réseau ou du journal, puisque ces incidents deviennent probables dans cette démarche.

3. Concevoir un système résilient

La redondance doit éviter les causes communes : deux instances dans le même domaine de panne ne suffisent pas. Les délais et reprises bornés empêchent une dépendance lente de bloquer tout le service. L'idempotence prévient les doubles opérations, un coupe-circuit réduit l'appel d'une dépendance défaillante, et les files permettent de reprendre un travail identifié. Des ressources séparées et un mode dégradé peuvent préserver une fonction essentielle.

Par exemple, la perte d'une instance de calcul peut être compensée par une autre à partir d'une tâche durable, tandis qu'un débit réseau réduit appelle une file bornée et un ralentissement des producteurs. Les sauvegardes doivent être restaurables et les mesures doivent vérifier le retour à un service correct. La résilience se démontre par ces comportements observés, pas par le seul nombre de machines.

Références techniques

[Linux, fichiers proc](#), [Python 3.8, exécution concurrente](#), [Microsoft SEAL, chiffrement homomorphe](#).