TP 2 — Mise en place d'un cluster Redis et benchmark de performance

Prérequis : Docker et Docker Compose installés, un terminal, redis-cli disponible localement (ou via un conteneur). Durée indicative : 3 à 4 heures.

Objectifs pédagogiques

- Déployer un cluster Redis à 6 nœuds (3 maîtres + 3 répliques) avec Docker.
- Comprendre le sharding par hash slots et vérifier la répartition des clés.
- Tester la haute disponibilité (failover) en tuant un nœud maître.
- Mesurer les performances avec redis-benchmark et interpréter les résultats.
- Comparer les performances entre une instance simple et un cluster.
- Superviser le cluster en temps réel avec Prometheus (collecte) et Grafana (visualisation).

Partie A — Mise en place d'un cluster Redis (6 nœuds)

A.1 Architecture cible

6 conteneurs Redis (3 maîtres, 3 répliques), chacun avec sa propre configuration, reliés sur un réseau Docker dédié. Chaque maître possède une réplique pour la haute disponibilité.

```
Maître 1 (7001) ── Réplique de 1 (7004)
Maître 2 (7002) ── Réplique de 2 (7005)
Maître 3 (7003) ── Réplique de 3 (7006)
```

A.2 Préparer l'arborescence

```
mkdir -p ~/tp-redis-cluster && cd ~/tp-redis-cluster
for port in 7001 7002 7003 7004 7005 7006; do
  mkdir -p ${port}/data
  cat > ${port}/redis.conf << EOF
port 6379
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
dir /data
EOF
done
```

Explication des directives clés :

- cluster-enabled yes : active le mode cluster sur ce nœud.
- cluster-node-timeout : délai (ms) avant qu'un nœud soit considéré indisponible — déclenche le failover.

•appendonly yes : persistance AOF activée (vu au module 6 du cours).

A.3 Fichier docker-compose.yml

```
version: "3.8"

services:
  redis-7001:
    image: redis:7-alpine
    command: redis-server /usr/local/etc/redis/redis.conf
    ports: ["7001:6379"]
    volumes:
      - ./7001/redis.conf:/usr/local/etc/redis/redis.conf
      - ./7001/data:/data
    networks: { redis-net: { ipv4_address: 172.30.0.11 } }

  redis-7002:
    image: redis:7-alpine
    command: redis-server /usr/local/etc/redis/redis.conf
    ports: ["7002:6379"]
    volumes:
      - ./7002/redis.conf:/usr/local/etc/redis/redis.conf
      - ./7002/data:/data
    networks: { redis-net: { ipv4_address: 172.30.0.12 } }

  redis-7003:
    image: redis:7-alpine
    command: redis-server /usr/local/etc/redis/redis.conf
    ports: ["7003:6379"]
    volumes:
      - ./7003/redis.conf:/usr/local/etc/redis/redis.conf
      - ./7003/data:/data
    networks: { redis-net: { ipv4_address: 172.30.0.13 } }

  redis-7004:
    image: redis:7-alpine
    command: redis-server /usr/local/etc/redis/redis.conf
    ports: ["7004:6379"]
    volumes:
      - ./7004/redis.conf:/usr/local/etc/redis/redis.conf
      - ./7004/data:/data
    networks: { redis-net: { ipv4_address: 172.30.0.14 } }

  redis-7005:
    image: redis:7-alpine
    command: redis-server /usr/local/etc/redis/redis.conf
    ports: ["7005:6379"]
    volumes:
      - ./7005/redis.conf:/usr/local/etc/redis/redis.conf
      - ./7005/data:/data
    networks: { redis-net: { ipv4_address: 172.30.0.15 } }

  redis-7006:
    image: redis:7-alpine
```

```
    command: redis-server /usr/local/etc/redis/redis.conf
    ports: ["7006:6379"]
    volumes:
      - ./7006/redis.conf:/usr/local/etc/redis/redis.conf
      - ./7006/data:/data
    networks: { redis-net: { ipv4_address: 172.30.0.16 } }

networks:
  redis-net:
    ipam:
      config:
        - subnet: 172.30.0.0/24
```

Adresses IP fixes utilisées pour simplifier la commande de création du cluster ci-dessous.

Ce fichier sera complété en partie C avec les services redis_exporter, prometheus et grafana — inutile de relancer docker compose up maintenant, on y reviendra après les parties A et B.

A.4 Démarrer les nœuds

```
docker compose up -d
docker compose ps          # vérifier que les 6 conteneurs tournent
```

A.5 Créer le cluster

```
docker run -it --rm --network tp-redis-cluster_redis-net redis:7-alpine \
  redis-cli --cluster create \
  172.30.0.11:6379 172.30.0.12:6379 172.30.0.13:6379 \
  172.30.0.14:6379 172.30.0.15:6379 172.30.0.16:6379 \
  --cluster-replicas 1
```

Redis propose une répartition des 16384 hash slots entre les 3 maîtres et l'association de chaque réplique à un maître. Taper yes pour confirmer.

Résultat attendu (extrait) :

```
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 172.30.0.14:6379 to 172.30.0.11:6379
...
[OK] All 16384 slots covered.
```

A.6 Vérifier l'état du cluster

```
docker exec -it tp-redis-cluster-redis-7001-1 redis-cli -c cluster info
docker exec -it tp-redis-cluster-redis-7001-1 redis-cli -c cluster nodes
```

cluster_state:ok confirme que les 16384 slots sont couverts.

A.7 Manipuler des clés en mode cluster

```
docker exec -it tp-redis-cluster-redis-7001-1 redis-cli -c
127.0.0.1:6379> SET utilisateur:1 "Alice"
-> Redirected to slot [X] located at 172.30.0.1Y:6379
OK
127.0.0.1:6379> SET utilisateur:2 "Bob"
-> Redirected to slot [Y] located at 172.30.0.1Z:6379
OK
127.0.0.1:6379> CLUSTER KEYSLOT utilisateur:1
```

L'option -c (cluster mode) suit automatiquement les redirections MOVED. Sans elle, le client recevrait une erreur MOVED explicite plutôt qu'une redirection transparente.

Point clé à observer : deux clés différentes atterrissent presque toujours sur des nœuds différents, car le hash slot dépend du nom de la clé. Pour forcer plusieurs clés sur le même nœud (utile pour des transactions multi-clés), on utilise les hash tags :

```
SET commande:{client42}:1 "..."
SET commande:{client42}:2 "..."
CLUSTER KEYSLOT commande:{client42}:1
CLUSTER KEYSLOT commande:{client42}:2   # -> même slot, car seul
"client42" est hashé
```

A.8 Tester le failover

```
# Identifier quel conteneur est actuellement maître du slot testé
docker exec -it tp-redis-cluster-redis-7001-1 redis-cli -c cluster
nodes | grep master

# Simuler une panne en arrêtant un maître
docker stop tp-redis-cluster-redis-7002-1

# Observer la promotion automatique de sa réplique
docker exec -it tp-redis-cluster-redis-7001-1 redis-cli -c cluster
nodes
```

Après quelques secondes (délai cluster-node-timeout), la réplique associée est promue maître. Les écritures sur les slots concernés reprennent automatiquement.

```
# Redémarrer le nœud arrêté : il revient en tant que réplique
docker start tp-redis-cluster-redis-7002-1
```

Question de compréhension : pourquoi un cluster à 3 maîtres sans réplique (--cluster-replicas 0) ne peut-il pas faire de failover automatique ? (Réponse attendue : sans réplique, il n'y a aucun nœud candidat pour reprendre les slots du maître tombé — le cluster perd la disponibilité en écriture sur ces slots jusqu'à intervention manuelle.)

Partie B — Benchmark de performance avec redis-benchmark

B.1 Benchmark de référence sur une instance simple

```
docker run -d --name redis-simple -p 6380:6379 redis:7-alpine
docker exec -it redis-simple redis-benchmark -q -n 100000 -c 50
```

- -n 100000 : nombre total de requêtes.
- -c 50 : nombre de connexions clientes concurrentes.
- -q : sortie condensée (une ligne par commande testée).

Sortie type :

```
PING_INLINE: 95238.10 requests per second
SET: 89285.71 requests per second
GET: 96153.85 requests per second
INCR: 90909.09 requests per second
LPUSH: 87719.30 requests per second
RPUSH: 88495.58 requests per second
SADD: 91743.12 requests per second
ZADD: 85470.09 requests per second
```

B.2 Isoler une commande et faire varier la charge de payload

```
# Impact de la taille des valeurs (option -d, en octets)
docker exec -it redis-simple redis-benchmark -q -t SET -n 50000 -d
100
docker exec -it redis-simple redis-benchmark -q -t SET -n 50000 -d
1000
docker exec -it redis-simple redis-benchmark -q -t SET -n 50000 -d
10000
```

À observer : le débit (requêtes/seconde) diminue quand la taille du payload augmente — le goulot devient la bande passante réseau/sérialisation plutôt que le traitement de la commande elle-même.

B.3 Effet du pipelining

```
docker exec -it redis-simple redis-benchmark -q -t SET,GET -n
100000 -P 1
docker exec -it redis-simple redis-benchmark -q -t SET,GET -n
100000 -P 16
docker exec -it redis-simple redis-benchmark -q -t SET,GET -n
100000 -P 64
```

-P regroupe plusieurs commandes par aller-retour réseau (rappel du module 11 du cours : le pipelining réduit la latence réseau cumulée). Le débit doit augmenter significativement entre -P 1 et -P 16, avec un gain marginal décroissant au-delà.

Consigner les résultats dans un tableau :

Pipelining (-P)	SET (req/s)	GET (req/s)
1		
16		
64		

B.4 Comparer instance simple vs cluster

```
# Contre le cluster (nœud d'entrée 7001)
docker run --rm --network tp-redis-cluster_redis-net redis:7-
alpine \
  redis-benchmark -q -h 172.30.0.11 -p 6379 --cluster -n 100000 -c
50
```

L'option --cluster indique à redis-benchmark de suivre les redirections MOVED, indispensable pour benchmarker correctement un cluster (sinon la majorité des requêtes échoueraient sur les mauvais nœuds).

Comparer :

Configuration	SET (req/s)	GET (req/s)
Instance simple		
Cluster (6 nœuds)		

Question d'analyse : le cluster est-il plus rapide par requête individuelle, ou apporte-t-il surtout un gain en débit global agrégé (plusieurs clients frappant des nœuds différents en parallèle) ? (Réponse attendue : la latence par requête individuelle n'est pas meilleure — parfois légèrement pire à cause du hop de redirection — mais le débit global agrégé monte avec le nombre de nœuds car la charge en écriture se répartit.)

B.5 Monitorer pendant le test

Dans un second terminal, pendant qu'un benchmark tourne :

```
docker exec -it redis-simple redis-cli --latency
docker exec -it redis-simple redis-cli --stat
docker exec -it redis-simple redis-cli INFO stats | grep
instantaneous_ops_per_sec
docker exec -it redis-simple redis-cli SLOWLOG GET 10
```

- --latency : latence moyenne en continu (ms).
- --stat : vue d'ensemble en direct (mémoire, clients connectés, ops/sec).
- SLOWLOG : recense les commandes ayant dépassé le seuil slowlog-log-slower-than (10 000 µs par défaut) — utile pour détecter les commandes coûteuses en production (rappel module 11 : éviter KEYS *).

Partie C — Supervision du cluster avec Prometheus et Grafana

C.1 Principe

Redis n'expose pas nativement un format de métriques compatible Prometheus : on utilise redis_exporter (projet oliver006/redis_exporter), un petit service qui interroge chaque instance Redis (via INFO) et republie ses statistiques au format Prometheus. Un exporter est déployé par nœud du cluster.

```
Redis 7001 → redis_exporter (9121) ┐
Redis 7002 → redis_exporter (9122) ├─→ Prometheus (scrape) → Grafana (dashboards)
Redis 7003 → redis_exporter (9123) │
Redis 7004..7006 → exporters 9124-9126 ┘
```

C.2 Compléter le docker-compose.yml

Ajouter ces services au fichier de la partie A (les 6 services redis-70xx restent inchangés) :

```
  exporter-7001:
    image: oliver006/redis_exporter:latest
    command: ["--redis.addr=redis://172.30.0.11:6379"]
    ports: ["9121:9121"]
    networks: { redis-net: {} }
    depends_on: [redis-7001]

  exporter-7002:
    image: oliver006/redis_exporter:latest
    command: ["--redis.addr=redis://172.30.0.12:6379"]
    ports: ["9122:9121"]
    networks: { redis-net: {} }
    depends_on: [redis-7002]

  exporter-7003:
    image: oliver006/redis_exporter:latest
    command: ["--redis.addr=redis://172.30.0.13:6379"]
    ports: ["9123:9121"]
    networks: { redis-net: {} }
    depends_on: [redis-7003]

  exporter-7004:
    image: oliver006/redis_exporter:latest
    command: ["--redis.addr=redis://172.30.0.14:6379"]
    ports: ["9124:9121"]
    networks: { redis-net: {} }
    depends_on: [redis-7004]

  exporter-7005:
    image: oliver006/redis_exporter:latest
    command: ["--redis.addr=redis://172.30.0.15:6379"]
    ports: ["9125:9121"]
    networks: { redis-net: {} }
```

```
    depends_on: [redis-7005]

  exporter-7006:
    image: oliver006/redis_exporter:latest
    command: ["--redis.addr=redis://172.30.0.16:6379"]
    ports: ["9126:9121"]
    networks: { redis-net: {} }
    depends_on: [redis-7006]

  prometheus:
    image: prom/prometheus:latest
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus-data:/prometheus
    ports: ["9090:9090"]
    networks: { redis-net: {} }
    depends_on: [exporter-7001, exporter-7002, exporter-7003,
exporter-7004, exporter-7005, exporter-7006]

  grafana:
    image: grafana/grafana:latest
    ports: ["3000:3000"]
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
      - GF_AUTH_ANONYMOUS_ENABLED=false
    volumes:
      - ./grafana/provisioning:/etc/grafana/provisioning
      - grafana-data:/var/lib/grafana
    networks: { redis-net: {} }
    depends_on: [prometheus]

volumes:
  prometheus-data:
  grafana-data:
```

Pourquoi un exporter par nœud plutôt qu'un seul exporter
global ? Chaque redis_exporter ne collecte que l'instance ciblée par --redis.addr. Répliquer un exporter par nœud permet à Prometheus d'étiqueter (labels) chaque série de métriques par nœud d'origine, et donc de comparer maître/répliques dans un même graphique.

C.3 Configuration Prometheus

```
mkdir -p prometheus
cat > prometheus/prometheus.yml << 'EOF'
global:
  scrape_interval: 5s

scrape_configs:
  - job_name: "redis_cluster"
    static_configs:
      - targets:
          - "exporter-7001:9121"
          - "exporter-7002:9121"
          - "exporter-7003:9121"
```

```
        - "exporter-7004:9121"
        - "exporter-7005:9121"
        - "exporter-7006:9121"
      labels:
        cluster: "tp-redis-cluster"
EOF
```

scrape_interval: 5s : Prometheus interroge chaque exporter toutes les 5 secondes — assez rapide pour visualiser un benchmark en direct sans surcharger les nœuds.

C.4 Provisioning Grafana (datasource + dashboard automatiques)

Pour éviter de configurer la datasource et d'importer le dashboard manuellement à chaque redémarrage :

```
mkdir -p grafana/provisioning/datasources
grafana/provisioning/dashboards
```

```
cat > grafana/provisioning/datasources/prometheus.yml << 'EOF'
apiVersion: 1
datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    url: http://prometheus:9090
    isDefault: true
EOF
```

```
cat > grafana/provisioning/dashboards/dashboards.yml << 'EOF'
apiVersion: 1
providers:
  - name: "Redis"
    folder: "Redis"
    type: file
    options:
      path: /etc/grafana/provisioning/dashboards
EOF
```

Télécharger un dashboard communautaire prêt à l'emploi pour redis_exporter (référence Grafana.com ID 763 — "Redis Dashboard for Prometheus Redis Exporter") :

```
curl -s
https://grafana.com/api/dashboards/763/revisions/latest/download \
  -o grafana/provisioning/dashboards/redis-dashboard.json
```

C.5 Démarrer la stack complète

```
docker compose up -d
docker compose ps    # 6 Redis + 6 exporters + prometheus +
grafana = 14 conteneurs
```

- Prometheus : http://localhost:9090 — onglet Status → Targets : les 6 exporters doivent apparaître à l'état UP.

•Grafana : http://localhost:3000 — connexion admin / admin (mot de passe défini dans le compose). Le dashboard "Redis" doit apparaître directement dans le dossier Redis grâce au provisioning.

C.6 Vérifier la collecte de métriques

Dans Prometheus (http://localhost:9090), onglet Graph, tester ces requêtes PromQL :

```
redis_up
redis_connected_clients
redis_memory_used_bytes
rate(redis_commands_processed_total[1m])
redis_keyspace_hits_total / (redis_keyspace_hits_total +
redis_keyspace_misses_total)
```

Métrique	Ce qu'elle indique
`redis_up`	1 si l'exporter arrive à joindre le nœud, 0 sinon (détecte une panne)
`redis_connected_clients`	Nombre de connexions clientes actives par nœud
`redis_memory_used_bytes`	Mémoire utilisée — à surveiller face à `maxmemory` (module 5 du cours)
`rate(redis_commands_processed_total[1m])`	Débit de commandes par seconde, en fenêtre glissante
Ratio hits/(hits+misses)	Taux de succès du cache — pertinent pour le cas d'usage « cache applicatif » (module 12)

C.7 Observer un benchmark en direct dans Grafana

1.Ouvrir le dashboard Grafana et le laisser affiché.
2.Dans un terminal, relancer le benchmark de la partie B contre le cluster :

docker run --rm --network tp-redis-cluster_redis-net redis:7-alpine \ redis-benchmark -q -h 172.30.0.11 -p 6379 --cluster -n 500000 -c 100

3.Observer en direct dans Grafana : le pic de commands_processed/sec, la montée de connected_clients, et la répartition de charge entre les 3 maîtres (un panneau par nœud si le dashboard 763 est utilisé).

Point clé à observer : la charge ne se répartit pas forcément de façon parfaitement égale entre les 3 maîtres — cela dépend de la distribution des clés générées par redis-benchmark sur les hash slots. C'est une bonne illustration concrète du sharding vu en A.7.

C.8 Provoquer une alerte visuelle : failover sous supervision

Reproduire le test de failover de la partie A.8 pendant que Grafana est ouvert :

```
docker stop tp-redis-cluster-redis-7002-1
```

Observer sur le dashboard : redis_up du nœud 7002 tombe à 0, pendant que sa réplique (métriques du nœud 7005) bascule de role=slave à role=master (visible via la métrique redis_instance_info ou le panneau dédié au rôle des nœuds selon le dashboard

utilisé). Cela relie visuellement le mécanisme de failover (module 9 du cours) à une preuve d'observabilité en production.

```
docker start tp-redis-cluster-redis-7002-1   # remise en service
```

Livrables attendus

1. Capture (ou copier-coller) de cluster info montrant cluster_state:ok.
2. Démonstration du failover : sortie de cluster nodes avant et après l'arrêt d'un maître.
3. Tableau rempli des résultats de benchmark (pipelining et comparaison simple/cluster).
4. Réponse argumentée aux deux questions de compréhension posées dans les parties A.7 et B.4.
5. Capture d'écran du dashboard Grafana montrant : (a) l'état UP des 6 nœuds, (b) le pic de débit pendant le benchmark de C.7, (c) la bascule redis_up observée lors du failover de C.8.

Pour aller plus loin (optionnel)

- Relancer le benchmark en modifiant maxmemory-policy (module 5 du cours) et observer l'effet une fois la mémoire saturée par de gros volumes de clés — visible dans Grafana via redis_memory_used_bytes et redis_evicted_keys_total.
- Ajouter un 4e maître au cluster à chaud avec redis-cli --cluster add-node puis rééquilibrer les slots avec --cluster reshard, en observant le rééquilibrage de charge dans Grafana.
- Refaire le test B.4 avec redis-benchmark -a <mot_de_passe> sur une instance protégée par requirepass (module 10 — sécurité), et mesurer l'overhead de l'authentification.
- Créer une règle d'alerte Prometheus (alerting_rules.yml) déclenchée quand redis_up == 0 pendant plus de 30 secondes, et brancher un Alertmanager pour une notification (email, Slack...).