Task 1
Реалізація каунтера з використанням Hazelcast

Запустив три консолі, в кожній hz start

```
Members {size:3, ver:3} [
        Member [127.0.0.1]:5701 - 3b52aaf4-14fd-4173-a94f-94aedcb6bc25 this
        Member [127.0.0.1]:5702 - 57ac536d-27c1-4ce3-b597-b9aa7c964756
        Member [127.0.0.1]:5703 - d893205b-1ada-4a33-8aa3-c26019c57e12
]
```

Результат:

```
---------------Performance Results----------------
No lock                  : 16537      | 52.751526 seconds
Pessimistic locking      : 100000     | 196.086184 seconds
Optimistic locking       : 100000     | 288.132398 seconds
--------------------------------------------------
```

```python
import hazelcast
import threading
import time

client = hazelcast.HazelcastClient()

def increment_counter(map_name):
    for _ in range(10000):
        value = map_name.get("counter")
        map_name.put("counter", value + 1)

def increment_counter_pessimistic(map_name):
    for _ in range(10000):
        map_name.lock("counter")
        try:
            value = map_name.get("counter")
            map_name.put("counter", value + 1)
        finally:
            map_name.unlock("counter")

def increment_counter_optimistic(map_name):
    for _ in range(10000):
        while True:
            old_value = map_name.get("counter")
            new_value = old_value + 1
```

```
            if map_name.replace_if_same("counter", old_value,
new_value):
                break

def run_threads(target_function):
    start_time = time.time()
    map_name = client.get_map("my-distributed-map").blocking()
    map_name.put("counter", 0)
    threads = []
    for _ in range(10):
        thread = threading.Thread(target=target_function,
args=(map_name,))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()
    end_time = time.time()
    diff_time = end_time - start_time
    return f"{map_name.get('counter')}", diff_time

def print_results(label, result_counter, diff_time):
    print(f"{label:25}: {result_counter:<10} | {diff_time:.6f}
seconds")

print("Performance Results".center(50, '-'))
print_results("No lock", *run_threads(increment_counter))
print_results("Pessimistic locking",
*run_threads(increment_counter_pessimistic))
print_results("Optimistic locking",
*run_threads(increment_counter_optimistic))
print("-" * 50)

client.shutdown()
```

У файлі /usr/lib/hazelcast/config/hazelcast.xml було змінено, щоб cp-member-count

```
    <cp-subsystem>
        <cp-member-count>3</cp-member-count>
        <missing-cp-member-auto-removal-seconds>14400</missing-cp-member-auto-removal-seconds>
    </cp-subsystem>
```

```
CP Group Members {groupId: METADATA(0), size:3, term:1, logIndex:0} [
        CPMember{uuid=9291afeb-7909-45f6-9f7a-5db3132acebf, address=[127.0.0.1]:5701} - LEADER
        CPMember{uuid=0200f81a-7a3f-4acc-9746-ca22d9dd26a2, address=[127.0.0.1]:5702}
        CPMember{uuid=0c39fda0-e37d-4016-ae1c-ef8bb398a73d, address=[127.0.0.1]:5703} - FOLLOWER this
]
```

Результати IAtomicLong

```
--------------------------------------------------
IAtomicLong
Final counter value: 100000
Execution time: 28.608991622924805 seconds
--------------------------------------------------
```

Код:

```python
import hazelcast
import threading
import time

client = hazelcast.HazelcastClient()

cp_subsystem = client.cp_subsystem
atomic_counter =
cp_subsystem.get_atomic_long("my-atomic-counter").blocking()

def increment_atomic_counter():
    for _ in range(10000):
        atomic_counter.increment_and_get()

threads = []
start_time = time.time()

for _ in range(10):
    thread = threading.Thread(target=increment_atomic_counter)
    threads.append(thread)
    thread.start()

for thread in threads:
```

```
    thread.join()

end_time = time.time()

final_counter_value = atomic_counter.get()
execution_time = end_time - start_time

print("\n" + "-" * 50)
print("IAtomicLong")
print(f"{'Final counter value:'} {final_counter_value}")
print(f"{'Execution time:'} {execution_time} seconds")
print("-" * 50)

client.shutdown()
```

Висновки:
Без блокувань підхід виявився найшвидшим, але ненадійним, оскільки кінцеве значення лічильника було некоректним через виникнення race condition. Песимістичне блокування забезпечує правильний результат, але значно уповільнює виконання через накладні витрати на блокування доступу до ресурсу. Оптимістичне блокування також дає правильний результат, але через часті невдалі спроби оновлення є ще повільнішим. Використання IAtomicLong з увімкненою підтримкою CP Subsystem забезпечує найкраще співвідношення продуктивності та надійності, гарантує правильний результат і відмовостійкість.