**## \*\*Parte 1 – Questões Teóricas\*\***

**\*\*1. Definição conceitual:\*\***

Sistema distribuído = conjunto de computadores independentes que se apresentam ao usuário como um sistema único e coerente.

Exemplos: Google Drive, Netflix, Blockchain, WhatsApp (na verdade são clusters conectados)

Grid computacional com máquinas heterogêneas - hardware e SO diferentes. Isso gera, maior dificuldade de manutenção, contudo gera maior tolerância a falhas. Suscetível a ataques cibernéticos. Baseado no modelo TCP/IP: endereço ip, porta lógica, socket, máscara de rede, protocolos tcp ou udp.

**\*\*2. Objetivos principais:\*\***

\* Compartilhamento de recursos: cpu, ram e disco

\* Transparência (de acesso, localização, replicação, falha).

\* Confiabilidade.

\* Escalabilidade.

**\*\*3. Comunicação síncrona vs assíncrona:\*\***

\* **\*\*Síncrona:\*\*** remetente espera resposta (bloqueante). → Mais simples, mas menos eficiente.

\* **\*\*Assíncrona:\*\*** remetente não espera (não bloqueante). → Mais rápido, mas exige controle. Isso só consegue acontecer devido ao uso da programação THREAD

**\*\*4. Arquiteturas:\*\***

\* **\*\*Cliente-Servidor:\*\*** um servidor central atende vários clientes. Ex: Gmail. Em cluster ou em grid

\* **\*\*P2P:\*\*** cada nó atua como cliente e servidor. Ex: BitTorrent. Em geral é em grid, mas pode ser cluster de grid

**\*\*5. Tipos de falhas:\*\***

\* Falhas de comunicação: perda de pacotes.

\* Falhas de processo: crash de servidor.

\* Falhas de hardware: queda de energia, disco corrompido.

Plano de contingência: backup, redundância, reenvio de pacotes (tcp)

**\*\*6. Relógios físicos vs lógicos:\*\***

\* **\*\*Físicos:\*\*** baseados em tempo real (NTP).

\* **\*\*Lógicos:\*\*** baseados em ordem causal de eventos (Lamport).

**\*\*7. Algoritmo de Lamport:\*\***

Cada processo mantém contador lógico.

\* Evento local: incrementa contador.

\* Mensagem enviada: inclui contador.

\* Ao receber: atualiza contador = max(local, recebido)+1.

  → Define ordem causal de eventos ($a \rightarrow b$).

**\*\*8. Threads com memória compartilhada vs mensagens:\*\***

\* **\*\*Memória compartilhada:\*\*** comunicação via variáveis comuns. Mais rápido, mas sujeito a corrida, deadlock, ...

    Gerenciar isso: semáforos ou monitores: syncronized e lock; exclusão mútua: lock; join

\* **\*\*Mensagens:\*\*** comunicação explícita. Mais seguro, mas mais lento.

**\*\*9. Escalabilidade:\*\***

Capacidade de suportar mais usuários/processos e hardware.

Técnicas:

\* Replicação/redundância de servidores.

\* Balanceamento de carga.

\* Particionamento de dados ou várias bases de dados redundantes.

**\*\*10. Caso Netflix:\*\***

\* Replicação/redundância de dados e equipamentos para disponibilidade.

\* Balanceamento de carga (CDNs).

\* Tolerância a falhas: redirecionamento automático em falhas de servidores.

**## \*\*Parte 2 – Questões Práticas em Python\*\***

**\*\*11. Thread básica (pares/ímpares):\*\***

```python

import threading

def pares():

    for i in range(0, 21, 2):

        print("Par:", i)

def impares():

    for i in range(1, 20, 2):

        print("Ímpar:", i)

t1 = threading.Thread(target=pares)

t2 = threading.Thread(target=impares)

t1.start()

t2.start()

t1.join()

t2.join()

```

---

**\*\*12. Threads com atraso:\*\***

```python

import threading, time

def tarefa(nome, atraso):

    for i in range(3):

        print(nome, "execução", i+1)

        time.sleep(atraso)

threads = [

    threading.Thread(target=tarefa, args=("Thread 1", 1000)),

    threading.Thread(target=tarefa, args=("Thread 2", 2000)),

    threading.Thread(target=tarefa, args=("Thread 3", 3000))

]

for t in threads: t.start()

for t in threads: t.join()

```

---

**\*\*13. Condição de corrida (sem lock):\*\***

```python

import threading

contador = 0

def incrementa():

    global contador

    for \_ in range(100000):

        contador += 1

t1 = threading.Thread(target=incrementa)

t2 = threading.Thread(target=incrementa)

t1.start();

t2.start()

t1.join();

t2.join()

print("Contador final:", contador)

# O valor varia, devido à condição de corrida

```

---

**\*\*14. Usando Lock:\*\***

```python

import threading

contador = 0

lock = threading.Lock()

def incrementa():

    global contador

    for \_ in range(100000):

        with lock:

            contador += 1

t1 = threading.Thread(target=incrementa)

t2 = threading.Thread(target=incrementa)

t1.start();

t2.start()

t1.join();

t2.join()

print("Contador final:", contador)

```

---

**\*\*15. Produtor-Consumidor:\*\***

```python

import threading, queue, time, random

fila = queue.Queue()

def produtor():

    for i in range(5):

        item = random.randint(1,100)

        print("Produzido:", item)

        fila.put(item)

        time.sleep(1)

def consumidor():

    for i in range(5):

        item = fila.get()

        print("Consumido:", item)

        time.sleep(2)

t1 = threading.Thread(target=produtor)

t2 = threading.Thread(target=consumidor)

t1.start();

t2.start()

t1.join();

t2.join()

```

---

**\*\*16. Relógio lógico (Lamport simplificado):\*\***

```python

import threading, time, random

contador = [0, 0]

def processo(id, outro):

    global contador

    for \_ in range(5):

        contador[id] += 1

        print(f"Processo {id}: evento local, clock {contador[id]}")

        if random.random() > 0.5:

            contador[outro] = max(contador[id], contador[outro]) + 1

            print(f"Processo {id} envia msg → Processo {outro}, clock {contador}")

        time.sleep(1)

t1 = threading.Thread(target=processo, args=(0,1))

t2 = threading.Thread(target=processo, args=(1,0))

t1.start();

t2.start()

t1.join();

t2.join()

```

---

**\*\*17. Thread com parâmetro (soma de lista):\*\***

```python

import threading

resultado = 0

def soma\_lista(lista):

    global resultado

    resultado = sum(lista)

lista = [1,2,3,4,5]

t = threading.Thread(target=soma\_lista, args=(lista,))

t.start()

t.join()

print("Soma:", resultado)

```

---

**\*\*18. Barrier (sincronização):\*\***

```python

import threading, time

barreira = threading.Barrier(4)

def tarefa(id):

    print(f"Thread {id} executando parte 1")

    time.sleep(1)

    barreira.wait()

    print(f"Thread {id} passou para parte 2")

# threads = [threading.Thread(target=tarefa, args=(i,)) for i in range(4)]

threads = []

for i in range(4):

    threads.append(threading.Thread(target=tarefa, args=(i,)))

for t in threads: t.start()

for t in threads: t.join()

```

---

**\*\*19. Deadlock e solução:\*\***

```python

import threading, time

lock1 = threading.Lock()

lock2 = threading.Lock()

def tarefa1():

    with lock1:

        time.sleep(1000)

        with lock2:

            print("Tarefa1 finalizada")

def tarefa2():

    with lock2:

        time.sleep(1000)

        with lock1:

            print("Tarefa2 finalizada")

t1 = threading.Thread(target=tarefa1)

t2 = threading.Thread(target=tarefa2)

t1.start(); t2.start()

t1.join(); t2.join()

# Deadlock pode ocorrer!

# Para evitar: adquirir locks em ordem consistente (sempre lock1 antes de lock2).

```