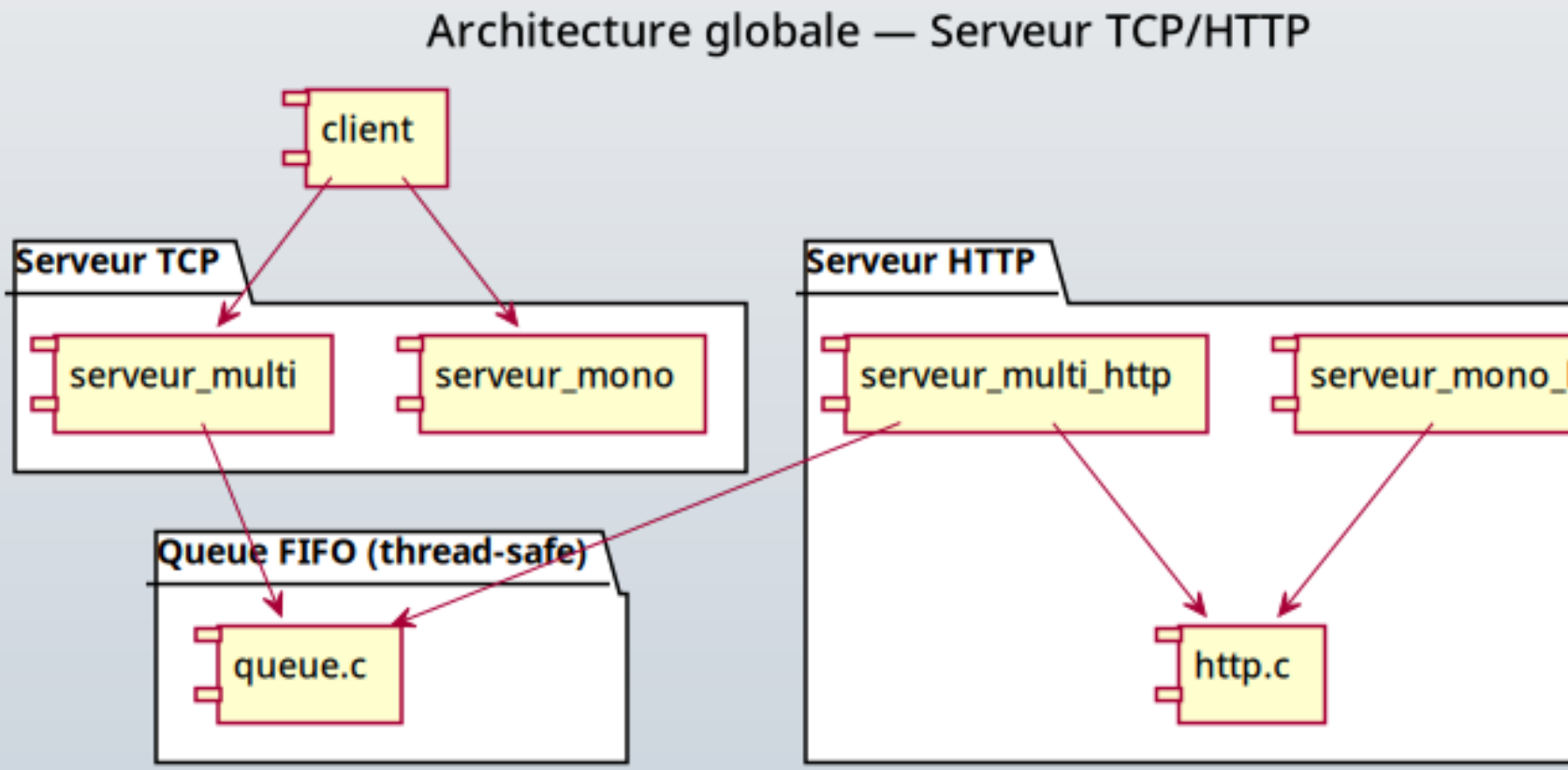
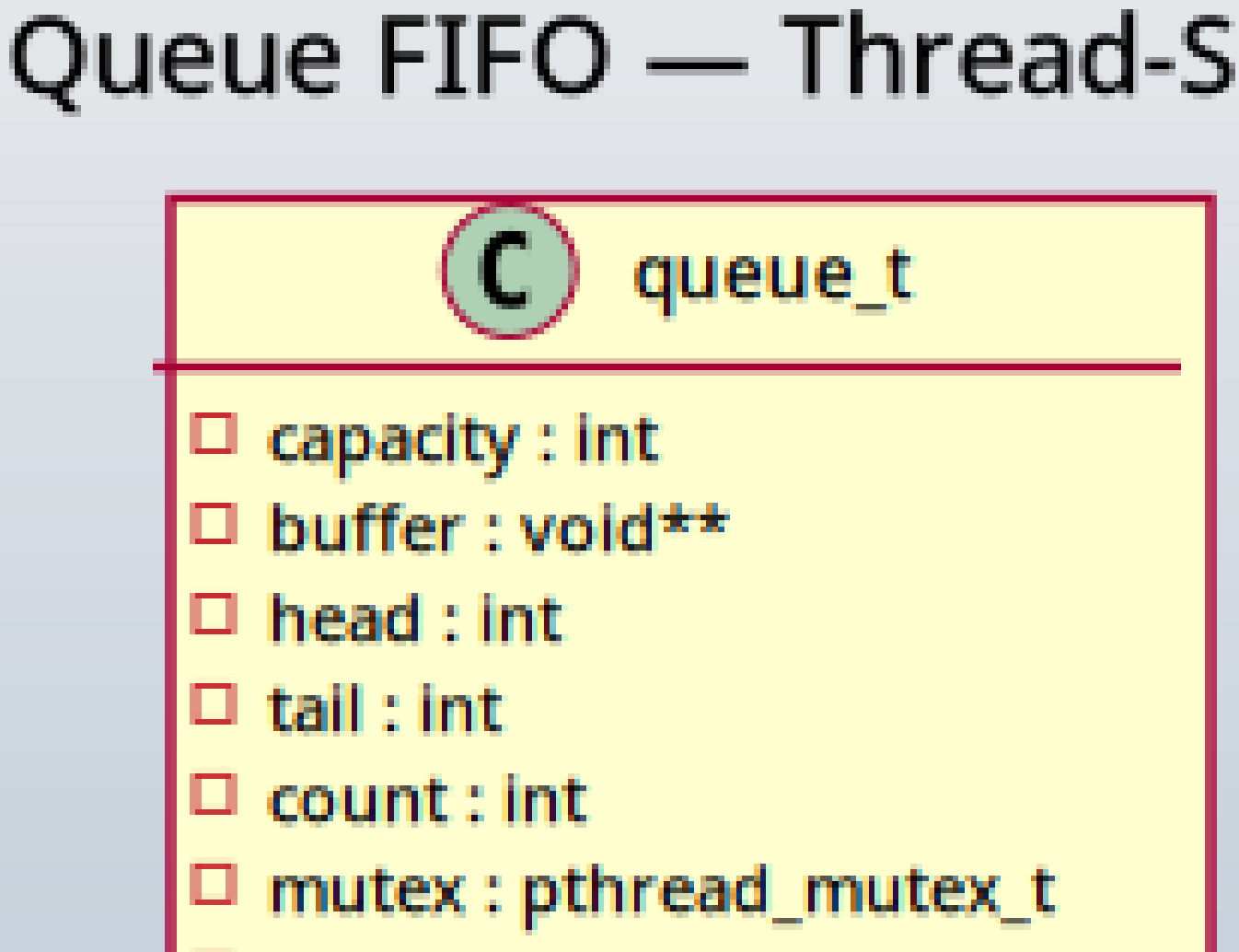
Serveurs TCP & HTTP Haute Performance

Multi-thread | Queue FIFO | Benchmarks | C/POSIX

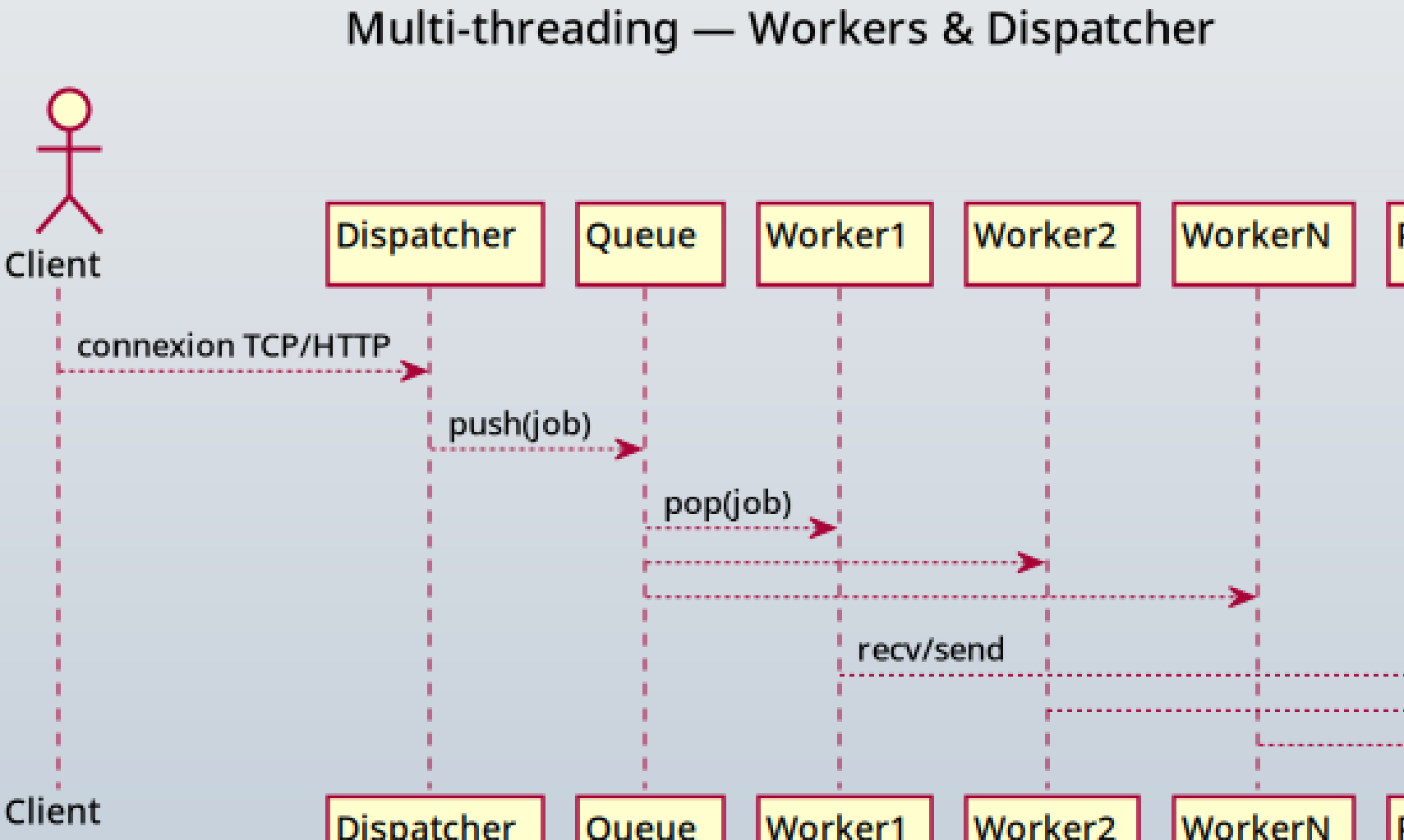
Architecture Globale



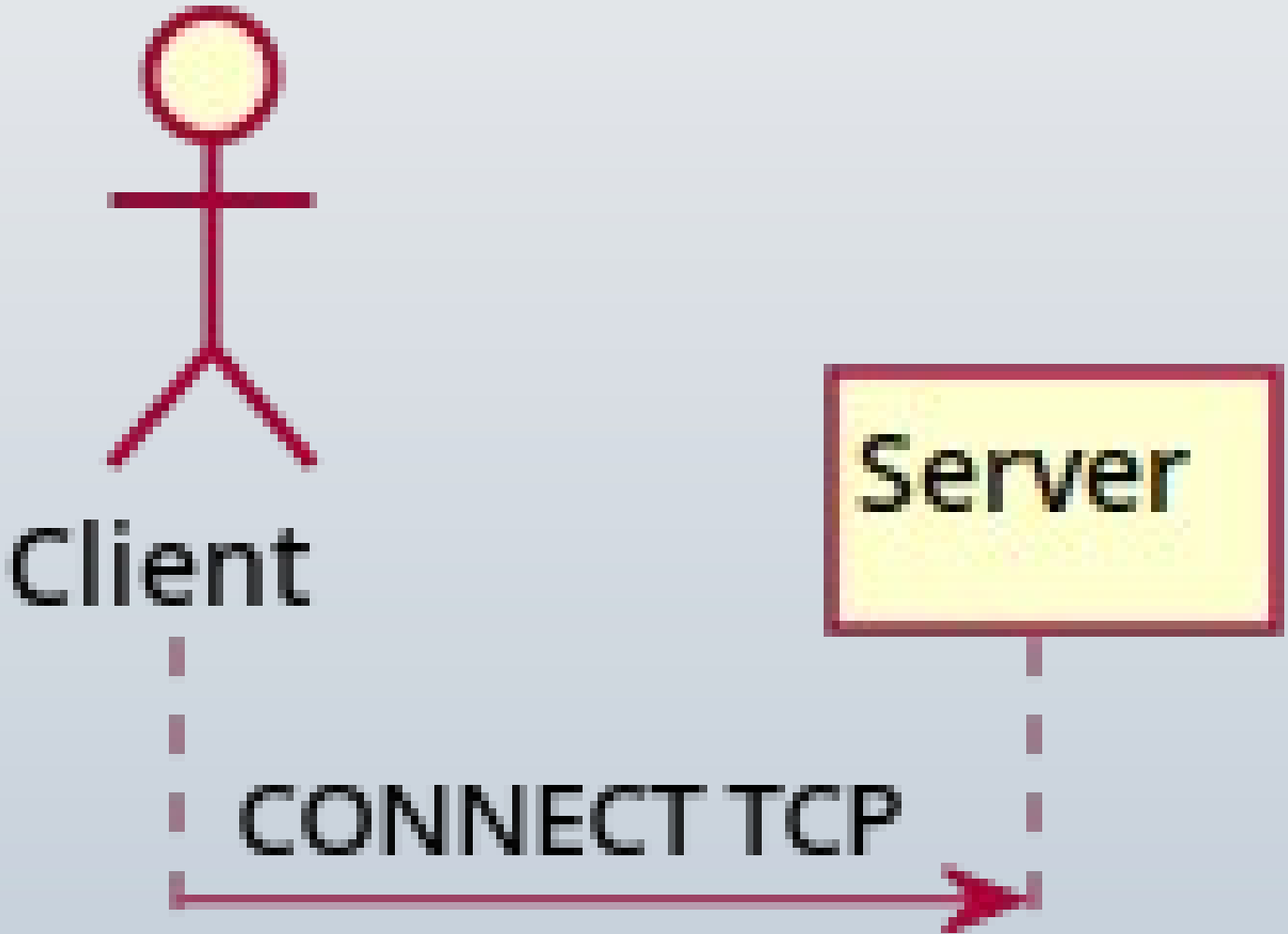
Queue FIFO Thread-Safe



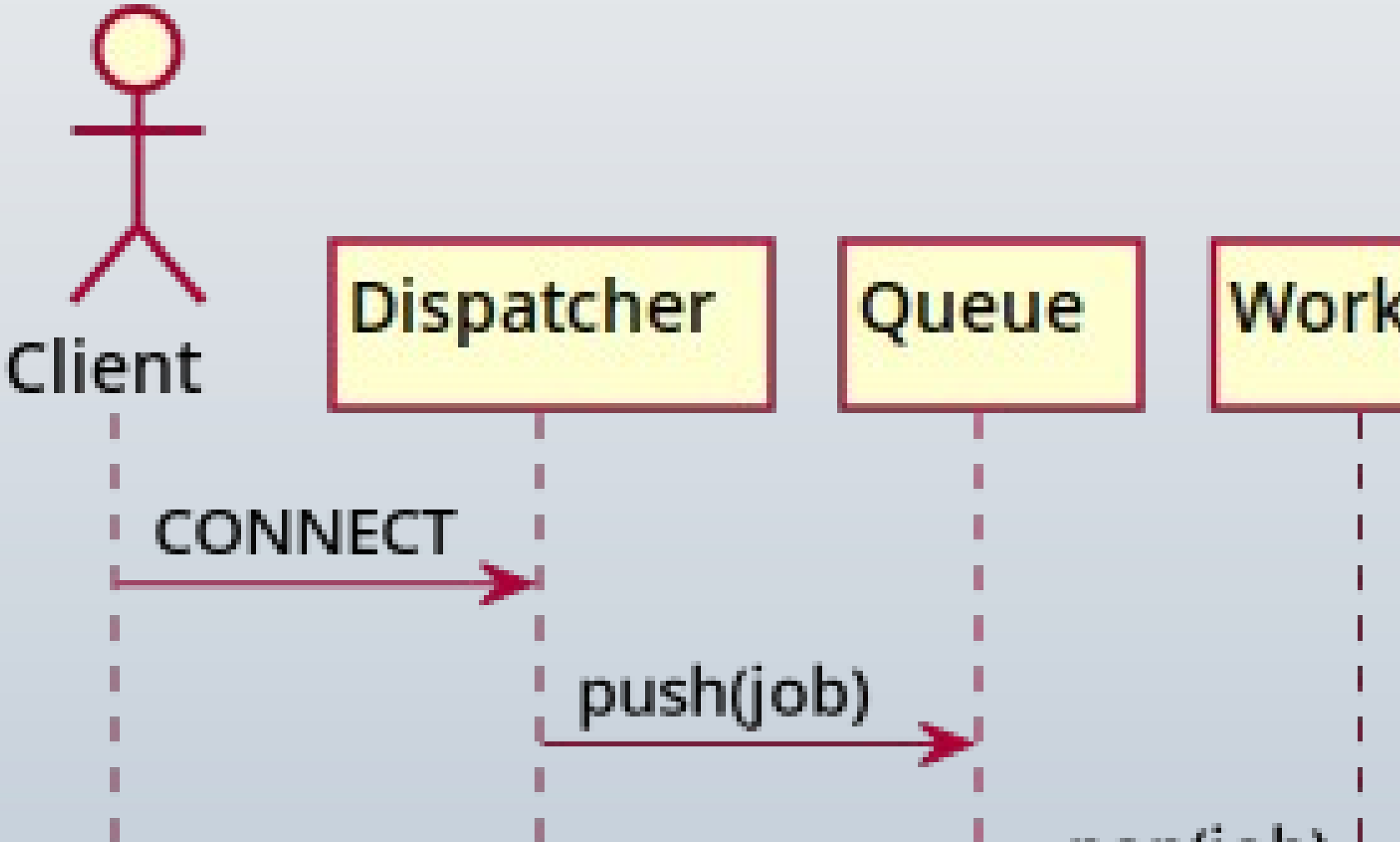
Threads & Workers



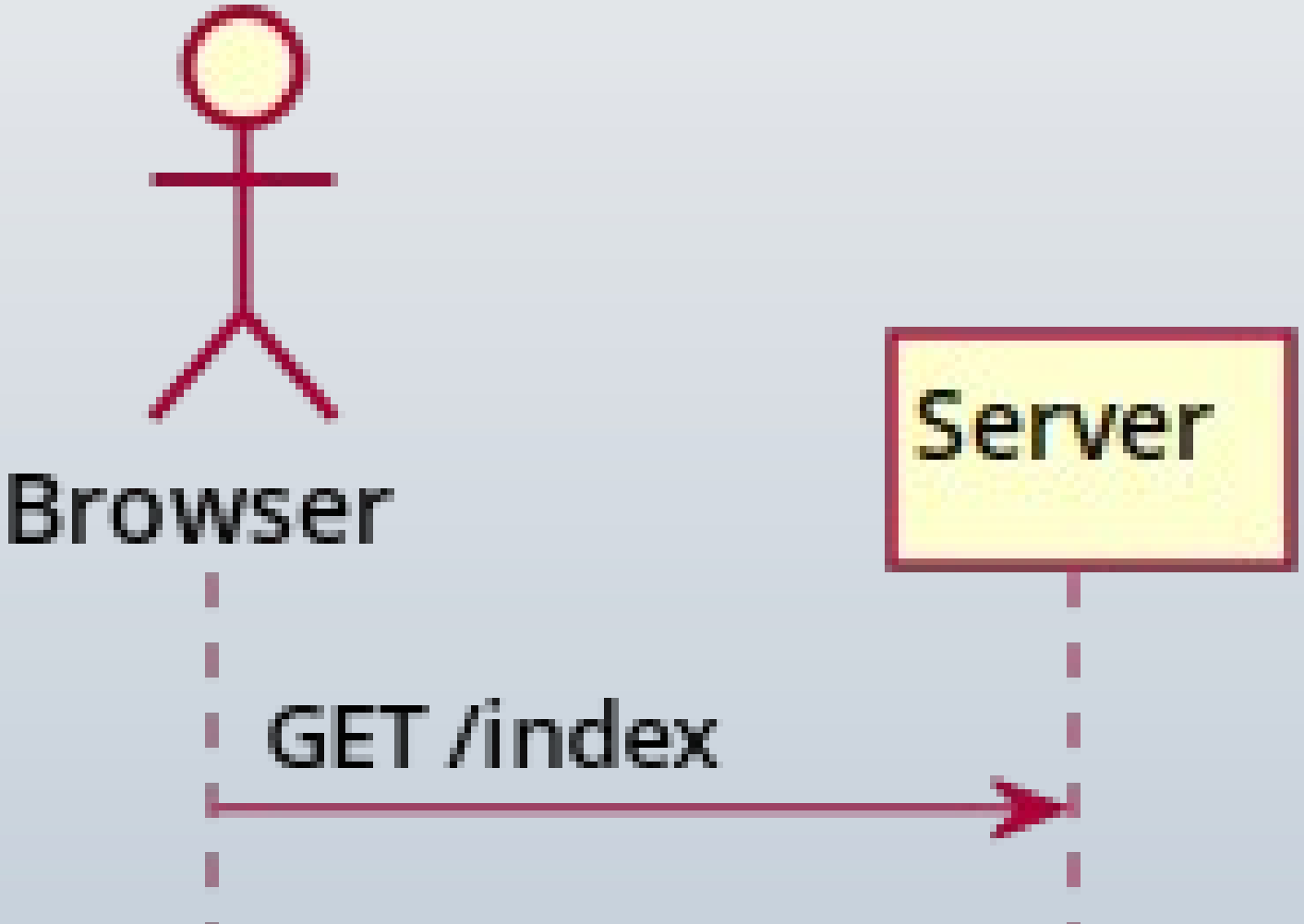
Séquence TCP Mono-thread



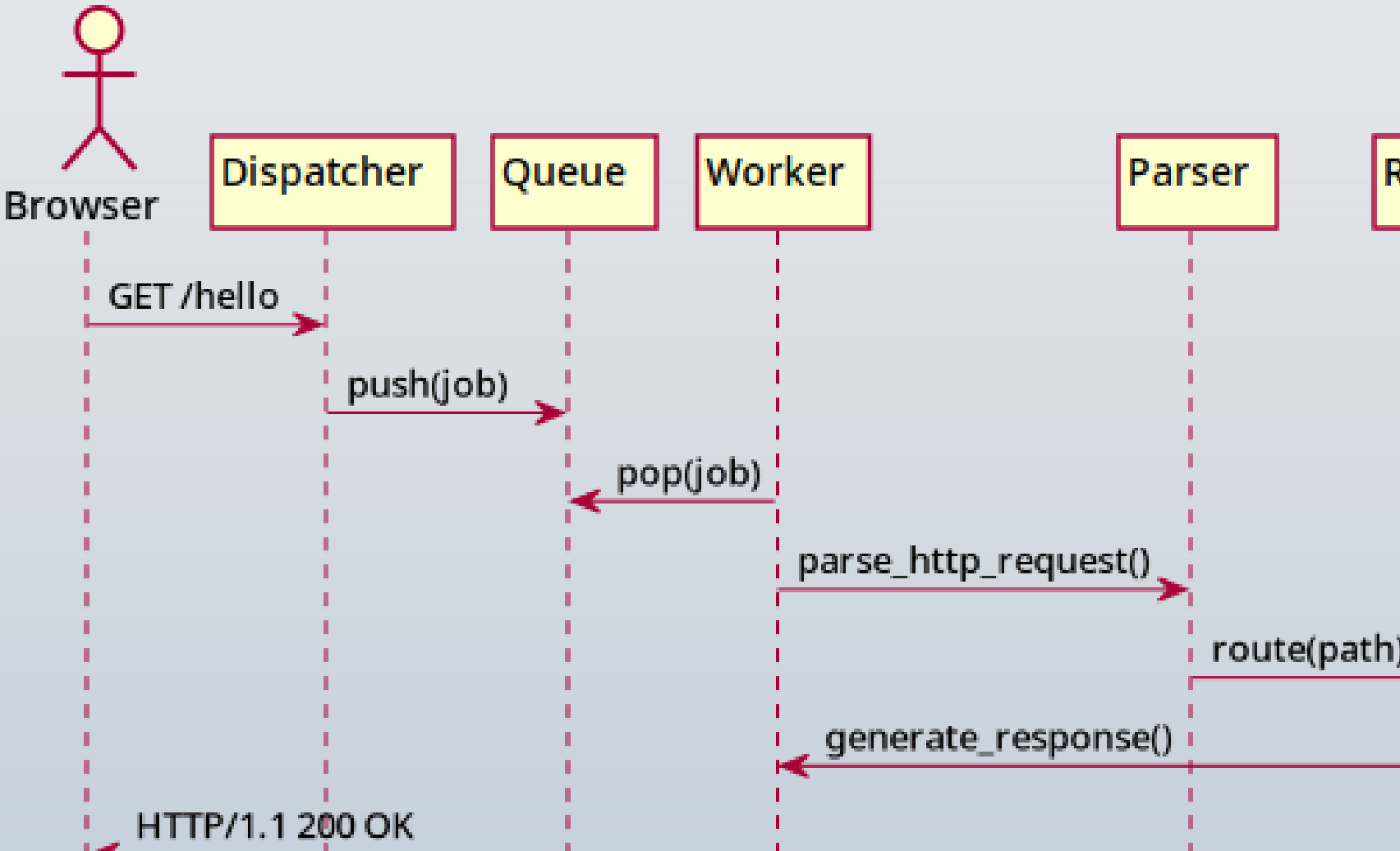
Séquence TCP Multi-thread



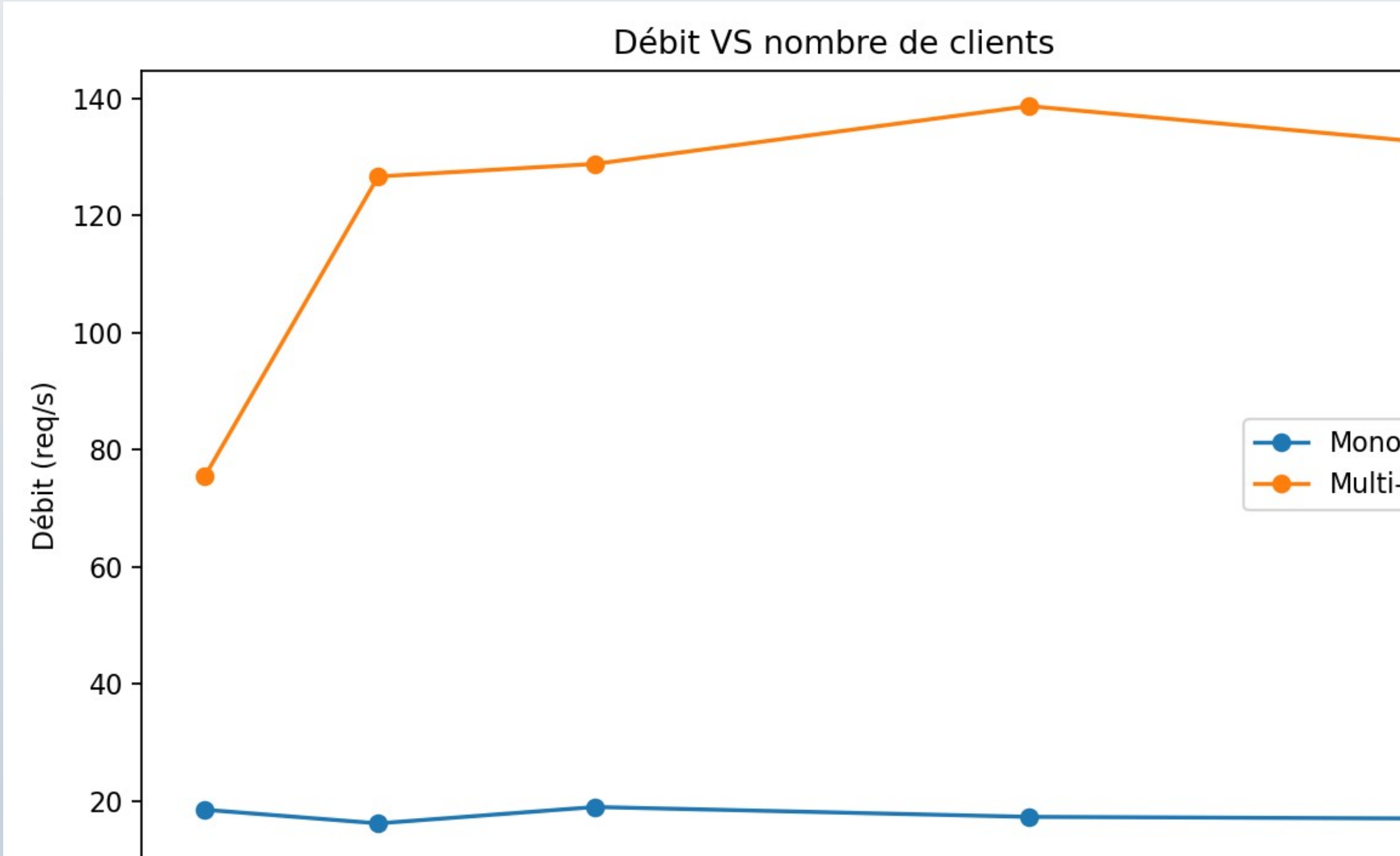
Séquence HTTP Mono-thread



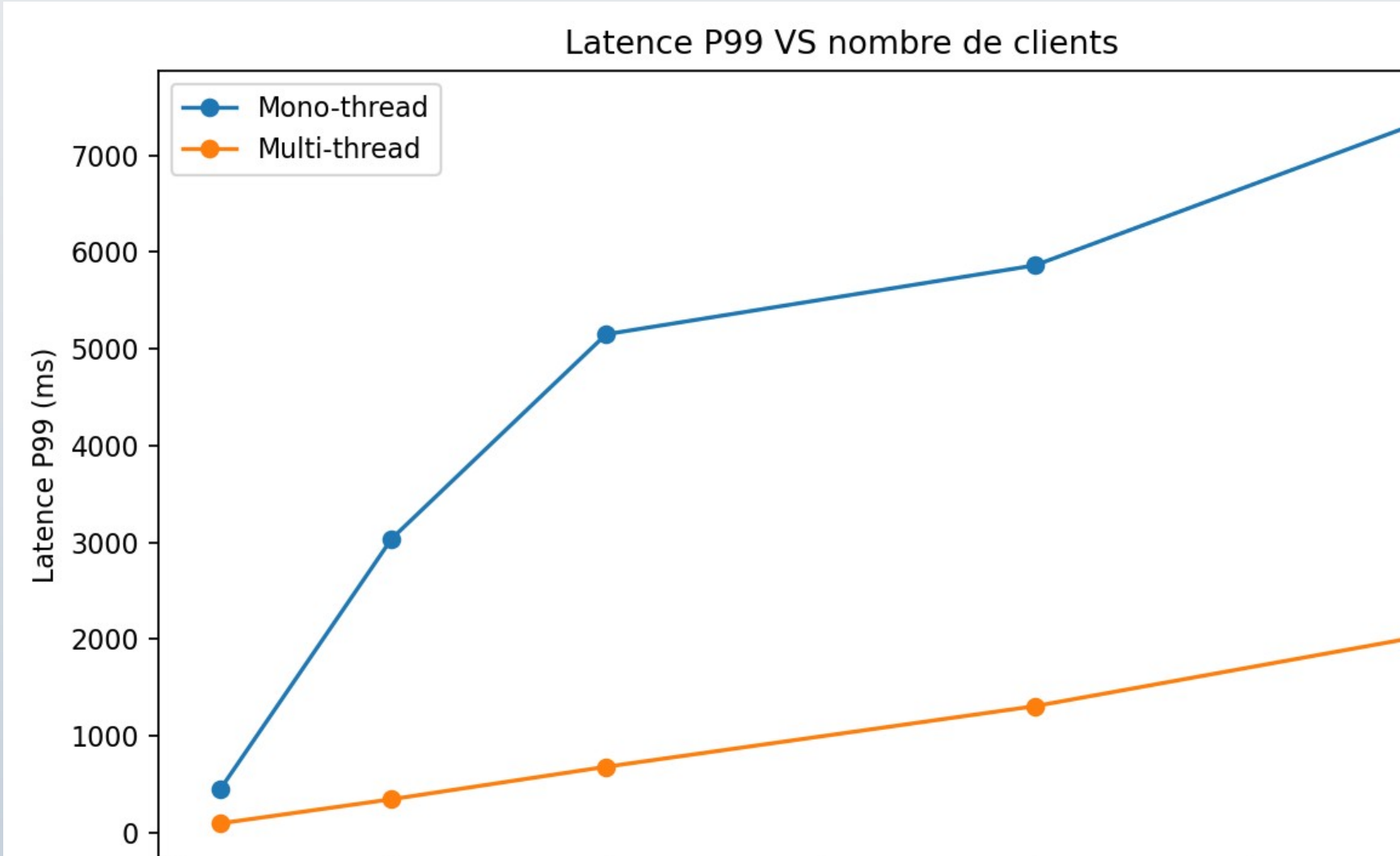
Séquence HTTP Multi-thread



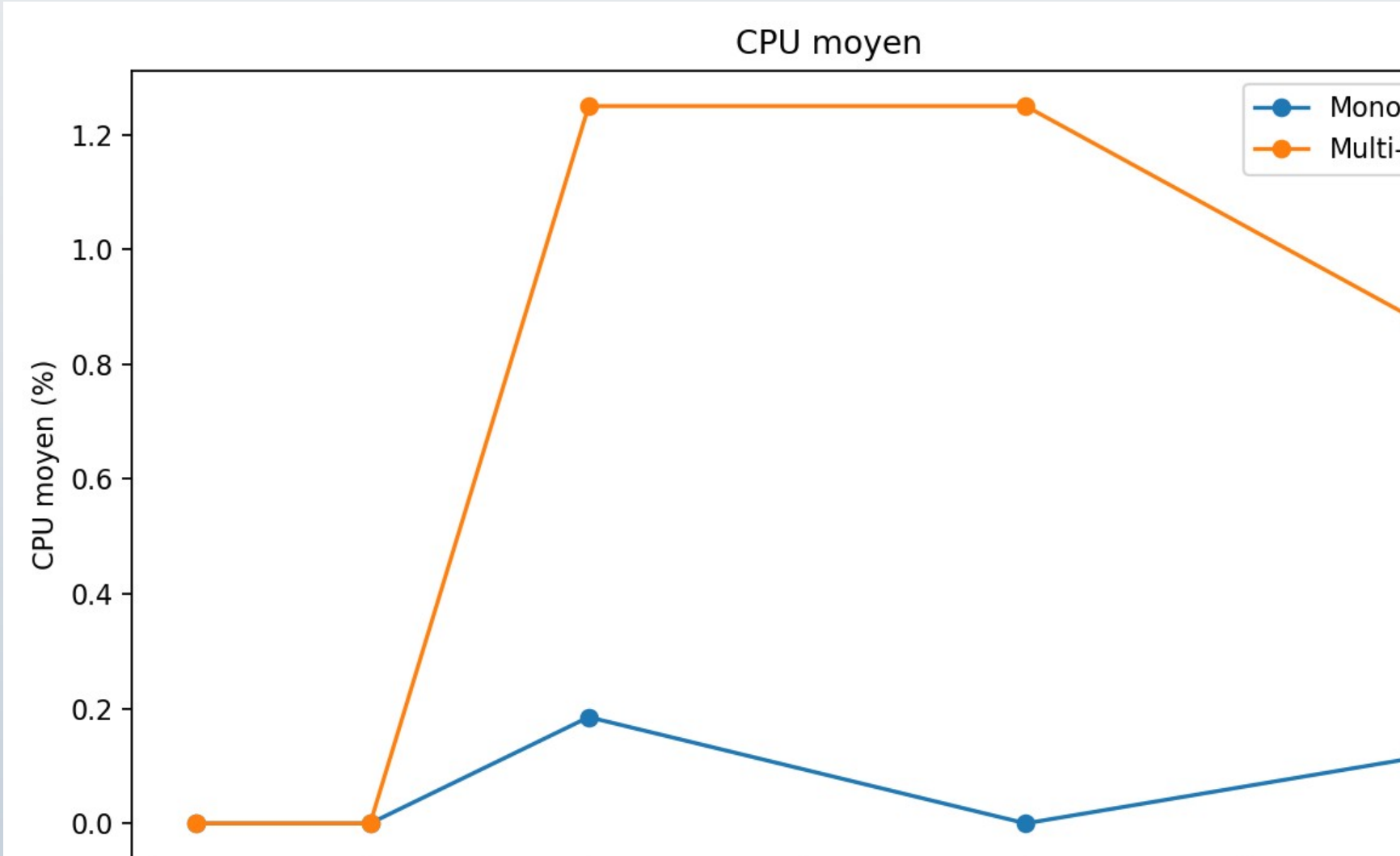
Throughput (req/s)



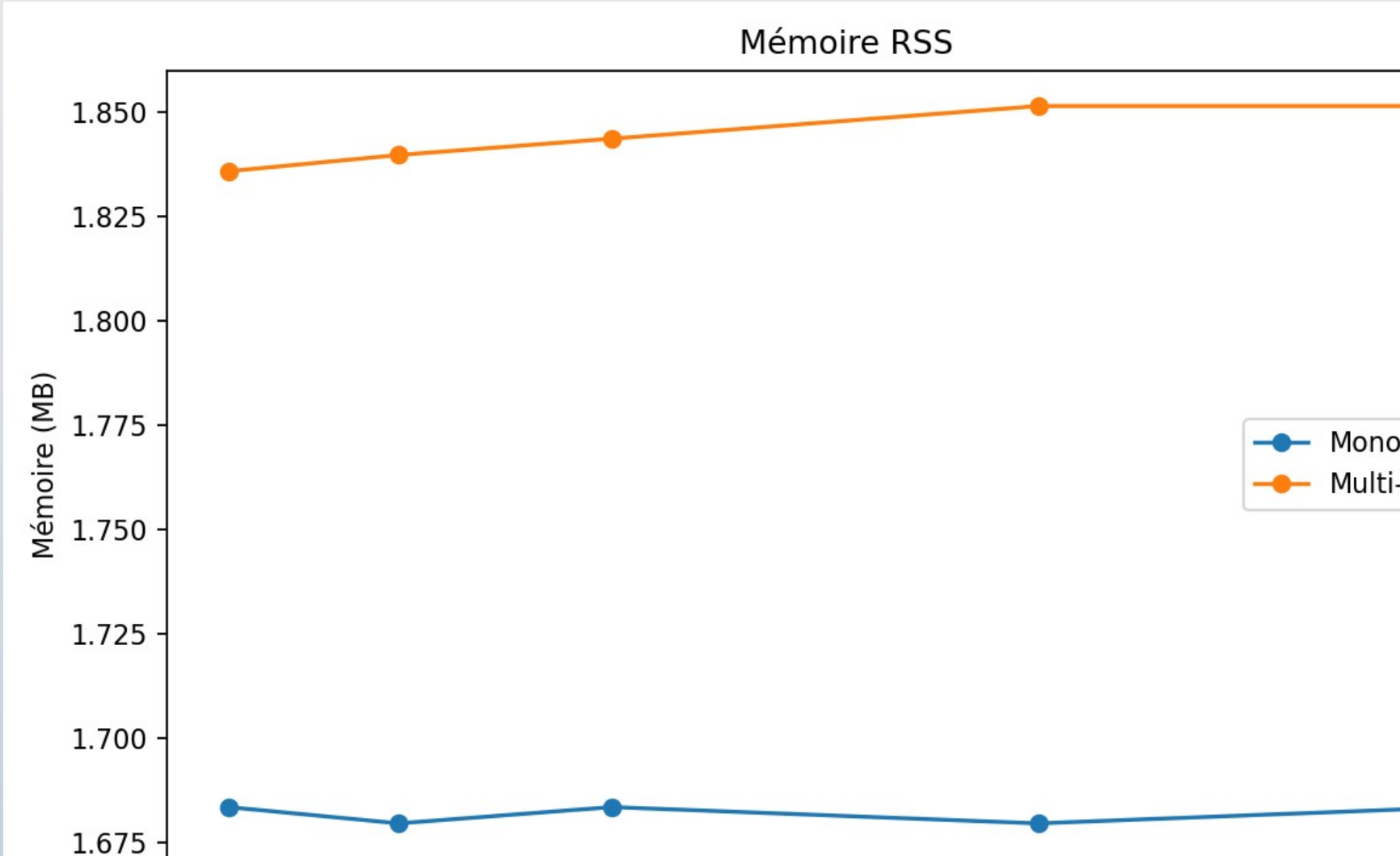
Latence P99 (µs)



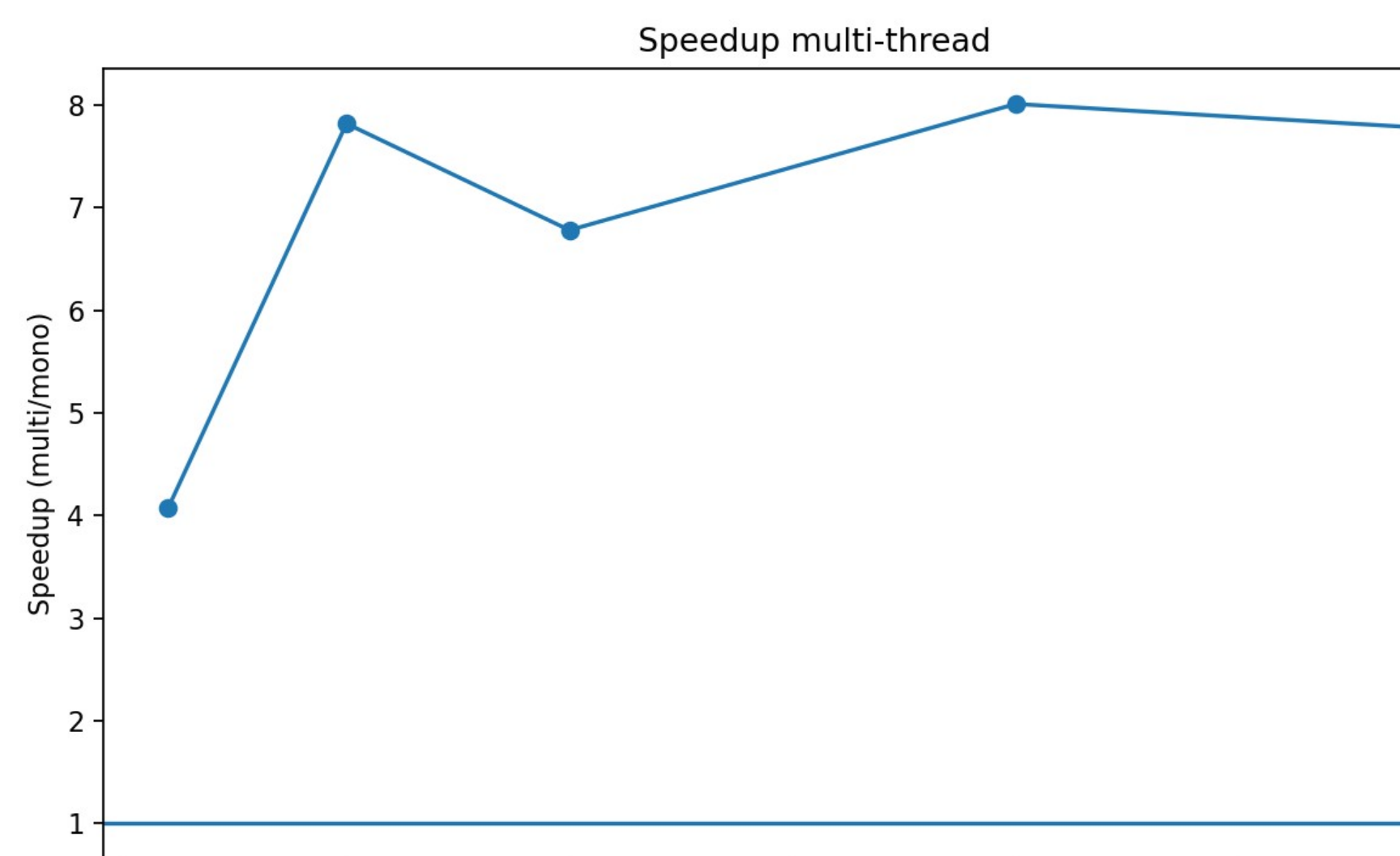
Utilisation CPU



Mémoire


Mémoire RSS
Mémoire (MB)
1.850
1.825
1.800
1.775
1.750
1.725
1.700
1.675
Mono
Multi-

Speedup Multi-thread



HTTP – Parser & Réponses (http.c)

Implémente le parsing de la ligne de requête HTTP (méthode, chemin, query).
Gère un découpage robuste des espaces et des paramètres après '?'.
Fournit une API simple pour les serveurs : parse_http_request() + send_http_response().
Encapsule la construction d'une réponse HTTP 1.1 (status line, headers, body).

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include "http.h"

void parse_http_request(const char *req, char *method, char *path, char
    char line[1024] = {0};

    /* On récupère la première ligne : "GET /chemin?x=1 HTTP/1.1" */
    const char *end = strstr(req, "\r\n");
    if (end) {
        size_t len = end - req;
        if (len > sizeof(line) - 1) {
            len = sizeof(line) - 1;
        }
        memcpy(line, req, len);
        line[len] = '\0';
    } else {
        strncpy(line, req, sizeof(line) - 1);
```

HTTP – Interface & Constantes (http.h)

Expose les prototypes du parser et de l'émetteur de réponse HTTP.
Centralise les tailles de buffers et types utilisés côté HTTP.
Permet de partager le même moteur HTTP entre serveur mono et multi-thread.

```c
#ifndef HTTP_H
#define HTTP_H

/**
 * parse_http_request
 * ------------------
 * Extrait la méthode, le chemin et la query string à partir d'une
 * requête HTTP brute.
 *
 * - req    : buffer contenant la requête brute
 * - method : buffer de sortie pour la méthode (GET, POST, ...)
 * - path   : buffer de sortie pour le chemin (/hello, /time, ...)
 * - query  : buffer de sortie pour la query (?a=1&b=2)
 */
void parse_http_request(const char *req, char *method, char *path, char

/**
 * send_http_response
 * ------------------
 * Envoie une réponse HTTP 1.1 complète :
```

Queue FIFO Thread-Safe (queue.c)

Implémente une file FIFO bornée, thread-safe, utilisée par le serveur multi-thread.
Utilise un mutex + 2 variables de condition (not_empty / not_full).
Supporte un mode shutdown propre pour réveiller tous les workers et le dispatcher.
Assure un comportement strictement FIFO et évite les conditions de course.

```c
#include "queue.h"
#include <stdlib.h>

void queue_init(queue_t *q, size_t size_max) {
    if (!q) return;

    q->head = q->tail = NULL;
    q->size = 0;
    q->size_max = size_max;  // 0 = illimité
    q->shutdown = false;

    pthread_mutexattr_t mutex_attr;
    if (pthread_mutexattr_init(&mutex_attr) != 0) {
        return;
    }

    if (pthread_mutexattr_settype(&mutex_attr, PTHREAD_MUTEX_ERRORCHECK)
        pthread_mutexattr_destroy(&mutex_attr);
        return;
    }
```

Queue FIFO – Interface (queue.h)

Définit la structure queue_t (head, tail, size, size_max, mutex, cond).
Expose queue_init(), queue_push(), queue_pop(), queue_shutdown(), queue_destroy().
Permet de réutiliser la même abstration pour TCP et HTTP (multi-thread).

```c
#ifndef QUEUE_H
#define QUEUE_H

#include <pthread.h>
#include <stdbool.h>
#include <stddef.h>

typedef struct queue_node {
    void *data;
    struct queue_node *next;
} queue_node_t;

/**
 * Queue FIFO thread-safe, bornée.
 * - mutex + condition variables not_empty / not_full
 * - shutdown permet de réveiller tous les threads en a
```

Serveur TCP Mono-thread (serveur_mono.c)

Boucle accept() → recv() → traitement_lourd() → send() pour un seul client à la fois.
Utilise un traitement CPU-bound simulé (~100ms) pour mesurer la saturation.
Renvoie le carré du nombre reçu (+ timestamp µs) au client.
SIGINT handler simple : fermeture du socket serveur et exit immédiat.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <signal.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <time.h>
#include <stdint.h>
#include <errno.h>

#define PORT 5050
#define BACKLOG 50   /* Amélioré pour éviter saturation */

/* Ignore SIGPIPE to handle broken connections gracefully */
#ifndef MSG_NOSIGNAL
#define MSG_NOSIGNAL 0
#endif
```

Serveur HTTP Mono-thread (serveur_mono_http.c)

Accepte les connexions une par une sur le port HTTP mono-thread (8080).
Parse la requête brute via http.c, route vers /, /hello, /time, /stats.
Gère des timeouts recv() pour éviter les connexions bloquées.
Idéal comme référence séquentielle pour comparer au multi-thread HTTP.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/time.h>

#include "http.h"

#define HTTP_PORT 8080
#define BACKLOG   32
#define BUF_SIZE  4096

/* Statistiques simples (non concurrentielles car mono-thread) */
static unsigned long total_requests   = 0;
static unsigned long hello_requests   = 0;
static unsigned long not_found_count  = 0;

static void route_request(int client_fd,
                          const char *method
```

Serveur TCP Multi-thread (serveur_multi.c)

Crée un pool fixe de WORKER_COUNT threads dès le démarrage.
Le thread principal accepte les connexions et les pousse dans la queue FIFO.
Chaque worker dépile un fd, exécute traitement_lourd(), renvoie la réponse, ferme le fd.
Gère SIGINT + queue_shutdown() pour un arrêt propre sans deadlock.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <signal.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <time.h>
#include <stdint.h>
#include <pthread.h>
#include <errno.h>

#include "queue.h"

/* Ignore SIGPIPE to handle broken connections gracefully */
#ifndef MSG_NOSIGNAL
#define MSG_NOSIGNAL 0
```

Serveur HTTP Multi-thread (serveur_multi_http

Architecture identique à TCP multi-thread, mais au niveau HTTP 1.1 (port 8081).
Workers parse la requête HTTP, appellent route_request(), renvoient une réponse JSON/HT
Statistiques globales /stats protégées par mutex (total_requests, hello_requests, 404).
Utilise SO_RCVTIMEO pour limiter la durée de blocage sur recv().

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <pthread.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/time.h>

#include "queue.h"
#include "http.h"

#define HTTP_PORT    8081        /* Port HTTP multi-thread */
#define BACKLOG      64
#define WORKERS      8
#define BUF_SIZE     4096

typedef struct {
    int client_fd;
} job_t;
```

Client de Charge / Benchmarks (python/client_stress.py)

Génère des centaines de clients concurrents pour mesurer throughput et latence.
Ouvre des connexions TCP/HTTP, envoie des requêtes, collecte les temps de réponse.
Produit des métriques agrégées (P50, P95, P99, RPS) en JSON / Excel.
Alimente le dashboard Plotly + les figures utilisées dans la présentation.

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Optimized TCP stress test client with better error handling and connection pooling.
"""

import socket
import struct
import time
import statistics
import sys
from typing import Dict, List, Any, Optional
from concurrent.futures import ThreadPoolExecutor, as_completed


def envoyer_requete(host: str, port: int, number: int) -> float:
    """Send single request and measure latency in milliseconds.

    Returns:
        float: Latency in ms, or -1.0 on error
    """
    start = time.perf_counter()
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            # Optimize socket options
            s.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)
```