Federico Navoni 5HT-I

TPSI

19 dicembre 2025

Elaborato Python BWT x socket

indice

premessa

Il progetto analizza lo sviluppo e la validazione di un sistema distribuito dedicato all'elaborazione della **Burrows-Wheeler Transform (BWT)**, un algoritmo di block-sorting fondamentale per la compressione dati e la bioinformatica. Il sistema è stato realizzato attraverso tre iterazioni incrementali: la **v0.1** ha introdotto una comunicazione socket TCP sincrona, la **v0.2** ha implementato il **multi-threading** e la persistenza JSON, mentre la **v0.3** ha completato l'architettura con un **bridge Flask** e la containerizzazione tramite **Docker**.

Ecco i pilastri tecnici emersi dalle fonti e dai test effettuati:

• **Algoritmo BWT**: La trasformazione si basa sull'ordinamento lessicografico delle rotazioni cicliche di una stringa, utilizzando il simbolo terminatore **$** per garantire l'unicità e la reversibilità del processo.

• **Infrastruttura di Rete**: È stato scelto il protocollo **TCP** per assicurare una consegna dei dati integra e ordinata, preferendo i socket binari diretti per minimizzare l'overhead rispetto alle astrazioni testuali come HTTP.

• **Isolamento e Persistenza**: L'uso di **Docker** e **Docker Compose** permette di isolare le risorse tramite Namespaces e Cgroups, mentre i **volumi** garantiscono che i dati in `output.json` persistano oltre il ciclo di vita dei container.

• **Sicurezza**: Una vulnerabilità critica è stata identificata nell'uso della libreria **Pickle**, che espone il sistema a rischi di **Remote Code Execution (RCE)** durante la deserializzazione di dati non fidati.

• **Risultati Empirici**: I test hanno confermato che il sistema è **thread-safe** grazie all'uso di `threading.Lock()` e che l'architettura a tre livelli introduce un overhead medio del **138,82%** nel passaggio da socket a HTTP.

1. Introduzione generale

Scopo e contesto del progetto

Il progetto ha come obiettivo centrale la progettazione e l'implementazione di un sistema distribuito per l'elaborazione della **Burrows-Wheeler Transform (BWT)**, un algoritmo di riorganizzazione dei dati fondamentale per la compressione lossless. Il contesto di sviluppo è stato definito da un requisito tecnologico stringente: **l'utilizzo dei socket come pilastro fondamentale per la comunicazione tra i componenti**. Mentre il calcolo della trasformata rappresenta la logica di centrale.

Breve introduzione alla Burrows-Wheeler Transform (BWT)

La BWT è una trasformazione reversibile che ordina lessicograficamente le **rotazioni cicliche** di una stringa, restituendo l'ultima colonna della matrice risultante. Un elemento tecnico vitale nel codice è l'uso del simbolo terminatore **$** (il carattere lessicograficamente più piccolo), indispensabile per garantire l'unicità della trasformazione e la sua completa **reversibilità** senza ambiguità.

Panoramica delle versioni sviluppate

Il sistema si è evoluto partendo dal nucleo dei socket e stratificando gli approfondimenti tecnici:

1. **v0.1 – Implementazione base**: Comunicazione socket TCP sincrona e bloccante tra client e server.
2. **v0.2 – Concorrenza e persistenza (Approfondimento)**: Introduzione del **multi-threading** e del salvataggio dei log su file `output.json`.
3. **v0.3 – Architettura avanzata (Approfondimento)**: Integrazione del **Bridge Flask** per l'interfaccia web e configurazione di **Docker Compose** con volumi per la persistenza dei dati sull'host.

2. Obiettivi del progetto

Il sistema è stato progettato per rispondere a una gerarchia di obiettivi che spaziano dalla corretta implementazione algoritmica alla robustezza dell'infrastruttura distribuita.

2.1 Obiettivo funzionale principale

• **Obiettivo Fondamentale (Core)**: Realizzare una connessione **Client-Server basata su socket TCP** che permetta l'invio di stringhe, la loro elaborazione remota tramite BWT e la ricezione del risultato.

2.2 Obiettivi secondari

Per elevare il software da semplice script a sistema distribuito di livello professionale, sono stati perseguiti i seguenti traguardi:

• **Misurazione delle prestazioni**: Integrare meccanismi di telemetria per calcolare il tempo di esecuzione (*elapsed time*) di ogni trasformazione. Questo permette di validare empiricamente la scalabilità dell'algoritmo rispetto alla complessità teorica O(n2logn).

• **Gestione di più client concorrenti**: Superare il limite del server bloccante tramite il **multi-threading**. Il sistema deve essere in grado di servire richieste simultanee, garantendo la sicurezza dei dati (*thread-safety*) tramite l'uso di meccanismi di sincronizzazione come `threading.Lock()`.

• **Persistenza dei dati**: Assicurare la tracciabilità delle operazioni attraverso il salvataggio sistematico di ogni elaborazione in un file **JSON**. I dati devono includere la stringa originale, il risultato BWT e i metadati temporali.

• **Accesso web tramite HTTP**: Fornire un'interfaccia utente moderna implementando un **Bridge Flask**. Questo componente agisce come trasformatore di protocollo, traducendo le richieste REST/JSON del browser in comandi socket TCP per il backend.

2.3 Obiettivi non funzionali

I requisiti strutturali che definiscono la qualità dell'architettura sono:

• **Modularità**: Separazione netta delle responsabilità tra il motore di calcolo (Backend Socket), il mediatore di protocollo (Flask Bridge) e l'interfaccia utente (Frontend HTML/JS).

• **Scalabilità**: Capacità del server di gestire un carico di lavoro crescente, sia in termini di lunghezza delle stringhe (efficienza algoritmica) che di numero di connessioni simultanee.

• **Affidabilità**: Garanzia di integrità dei dati sia durante la trasmissione (tramite TCP) che durante la scrittura su disco, prevenendo corruzioni o perdite di record in scenari di accesso concorrente.

• **Portabilità**: Utilizzo della containerizzazione tramite **Docker** per rendere il sistema indipendente dall'ambiente host. L'infrastruttura deve essere riproducibile tramite configurazioni "Infrastructure as Code" (Docker Compose) e garantire la persistenza dei dati sull'host tramite l'uso di volumi dedicati.

2.4 Approfondimenti

Approfondimenti e Miglioramenti: Al fine di elevare il progetto a uno standard professionale, sono stati integrati i seguenti moduli aggiuntivi:

- Persistenza dei dati: Salvataggio sistematico dei risultati in un file JSON.
- Gestione della Concorrenza: Implementazione di un sistema multi-thread protetto da lock per garantire l'integrità del file durante accessi simultanei.
- Containerizzazione: Utilizzo di Docker per isolare l'ambiente di esecuzione e garantire la portabilità tramite configurazioni "Infrastructure as Code".
- Accessibilità Web: Sviluppo di un front-end client basato su un Bridge Flask per permettere l'interazione tramite browser.

3. Tecnologie e strumenti utilizzati

In questa sezione vengono analizzati i pilastri tecnologici del progetto. La scelta degli strumenti è stata orientata alla creazione di un sistema che fosse, al contempo, un efficace banco di prova per la comunicazione a basso livello e un'architettura moderna e portabile.

3.1 Linguaggio di programmazione: Python

Il linguaggio **Python** (versione 3.11 nelle immagini Docker) è stato selezionato per la sua versatilità e per la ricchezza delle librerie dedicate alla manipolazione di stringhe e alla gestione di rete.

• **Motivazioni della scelta**: Python permette una prototipazione rapida di algoritmi complessi e offre una gestione semplificata dei socket rispetto a linguaggi come il C, pur mantenendo un controllo granulare.
• **Librerie standard utilizzate**:
  ◦ `socket`: per la gestione della comunicazione TCP/IP.
  ◦ `pickle`: per la serializzazione di oggetti Python (tuple contenenti risultati e tempi).
  ◦ `threading`: per implementare il server multi-thread e gestire la concorrenze.
  ◦ `json`: per la persistenza dei dati in formato leggibile e standardizzato.
  ◦ `os` e `time`: per la gestione del file system, dei volumi e la misurazione delle prestazioni.

3.2 Comunicazione di rete: Socket TCP

L'utilizzo dei **socket** è stato il requisito centrale della consegna per gestire lo scambio di dati tra client e server.

• **Motivazioni della scelta (TCP vs UDP)**: È stato scelto il protocollo **TCP** poiché garantisce l'**integrità**, l'**ordine** e l'**affidabilità** della consegna dei pacchetti tramite il meccanismo del *three-way handshake*. L'UDP, pur essendo più veloce per applicazioni real-time, è stato scartato in quanto "inaffidabile" (non garantisce che i dati arrivino o siano corretti), condizione inaccettabile per una trasformata algoritmica precisa come la BWT.

• **Encoding e Serializzazione**: I dati testuali sono stati codificati in **UTF-8** per supportare caratteri speciali ed emoji. Per lo scambio di metadati (risultati + tempi), è stata usata la libreria **Pickle**, che converte gli oggetti in flussi di byte. *Nota sulla sicurezza*: Sebbene efficace, l'uso di Pickle espone il sistema a rischi di **Remote Code Execution (RCE)** se utilizzato su dati non fidati, a causa della sua capacità di eseguire codice arbitrario durante la deserializzazione.

3.3 Algoritmi: Burrows-Wheeler Transform

L'algoritmo BWT riorganizza i caratteri di una stringa ordinando lessicograficamente le sue **rotazioni cicliche**.
• **Complessità computazionale**: Nel progetto è stata implementata la versione **naive**.
  ◦ **Temporale**: O(n2logn), dovuta alla generazione di n rotazioni e al loro ordinamento.
  ◦ **Spaziale**: O(n2), necessaria per memorizzare l'intera matrice delle rotazioni.

3.4 Tecnologie Web (v0.3): Flask e HTTP/REST

Nella versione v0.3 è stato introdotto il framework **Flask** per agire come **Bridge** tra il web e il backend socket. I socket TCP psono stati applicati solo per los scambio di dati fra client e server relativi alla gestione e alla codifica bwt, garantendo prestazioni superiori. I test hanno misurato un **overhead medio del 138,82%** passando i dati trasmessi da socket verso HTTP per l'interfaccia web, giustificato dalla comodità di un'interfaccia browser.

3.5 Containerizzazione (v0.3): Docker e Persistenza

Il server è stato interamente containerizzato per garantire l'isolamento e la riproducibilità dell'ambiente.
• **Docker e Docker Compose**: Docker isola le risorse tramite **Namespaces** (isolamento della vista) e **Cgroups** (limiti di CPU/Memoria). Docker Compose è stato utilizzato per orchestrare i servizi e mappare le porte (es. 65432) tra host e container.
• **Volumi per la persistenza**: Per assicurare che il file `output.json` non venisse perso alla chiusura dei container, è stato configurato un **volume bind mount** (`./data:/data`). Questo meccanismo permette al server di scrivere su un percorso che punta direttamente al filesystem dell'host, garantendo la sopravvivenza dei record di test.

4. Architettura generale del sistema

L'architettura del sistema è stata progettata seguendo il modello **client-server**, con un'evoluzione incrementale volta a migliorare la modularità e la gestione delle risorse. Il cuore ingegneristico risiede nella comunicazione a basso livello tramite **socket TCP**, che funge da strato di trasporto affidabile per i dati.

4.1 Architettura client-server

Il sistema si basa sulla separazione netta tra l'entità che richiede il servizio e quella che lo eroga.
• **Ruolo del Server**: Il server funge da nodo computazionale dedicato. Esso rimane in ascolto su una porta specifica (65432), riceve le stringhe dai client, applica l'algoritmo **BWT** e restituisce il risultato serializzato. Nelle versioni avanzate, il server gestisce anche la persistenza dei dati scrivendo i log su un file JSON.

• **Ruolo del Client**: Il client agisce come interfaccia per l'utente, inviando messaggi codificati in **UTF-8** e ricevendo risposte in formato **pickle** o **JSON**.

• **Separazione delle responsabilità**: L'architettura modulare garantisce che la logica di business (l'algoritmo BWT) risieda esclusivamente sul server, mentre il client si occupa solo della presentazione e della trasmissione dei dati.

4.2 Evoluzione architetturale

Il sistema ha attraversato tre fasi di maturazione tecnologica:

1. **Architettura monolitica (v0.1)**: Rappresenta l'implementazione base sincrona. In questa configurazione, il server è **bloccante**: può gestire un solo client alla volta e non può accettare nuove connessioni finché l'elaborazione corrente non è terminata. La comunicazione è binaria pura tramite socket TCP.

2. **Architettura multi-thread (v0.2)**: Introduzione della **concorrenza**. Grazie all'uso del modulo `threading` di Python, il server genera un nuovo thread per ogni connessione in entrata.

3. **Architettura a servizi (v0.3)**: L'evoluzione finale introduce un'architettura a **tre livelli**. Viene inserito un **Bridge Flask** che funge da intermediario. Questo bridge riceve richieste **HTTP/REST** dal browser e le traduce in comandi **Socket TCP** per il server backend. il sistema lato server è isolato tramite **Docker**, utilizzando i *Namespaces* per la vista delle risorse e i *Cgroups* per la limitazione dei consumi di CPU e memoria.In questa fase è stata implementata la **thread-safety** tramite `threading.Lock()` per evitare *race condition* durante il salvataggio dei risultati nel file `output.json`.

5. Algoritmo Burrows-Wheeler Transform (BWT)

La **Burrows-Wheeler Transform**, introdotta nel 1994, rappresenta una pietra miliare nel campo della compressione dati lossless e della bioinformatica. Sebbene non comprima i dati direttamente, li riorganizza in modo da facilitare la compressione successiva tramite tecniche come il *Run-Length Encoding* (RLE) o la codifica di Huffman.

5.1 Descrizione dell'algoritmo

L'algoritmo opera secondo una logica di **"block-sorting"**, elaborando blocchi di testo predefiniti per trasformarli in una permutazione reversibile.

1. **Concetto di rotazioni cicliche**: Data una stringa di input di lunghezza n, l'algoritmo genera tutte le sue N possibili **rotazioni cicliche** (o spostamenti circolari). Ad esempio, per la stringa "BANANA",lerotazioniincludono"BANANA", "ANANAB","NANABA", e così via.

2. **Ordinamento lessicografico**: Tutte le rotazioni generate vengono disposte in una matrice, chiamata **Burrows-Wheeler Matrix** (BWM), e ordinate alfabeticamente. Questo passaggio è fondamentale perché raggruppa le rotazioni che iniziano con prefissi simili.

3. **Estrazione dell'ultima colonna**: Il risultato della BWT è la stringa formata dai caratteri presenti nell'**ultima colonna** della matrice ordinata. Grazie all'ordinamento, caratteri che appaiono in contesti simili nel testo originale tendono a trovarsi vicini in questa colonna, creando il fenomeno del **"clustering"**.

5.2 Implementazione nel progetto

• **Analisi passo-passo del codice**:
  ◦ Viene aggiunta la sentinella alla stringa: `s = s + "$"`.
  ◦ Si generano le rotazioni tramite una *list comprehension*: `[s[i:] + s[:i] for i in range(n)]`.
  ◦ Si ordina la lista con il metodo `sort()` nativo di Python.
  ◦ Si estrae l'ultimo carattere di ogni riga e si unisce il tutto in una stringa finale: `"".join(r[-1] for r in rotazioni)`.

• **Scelta del simbolo terminatore $**: L'uso del carattere `$` è vitale per la **reversibilità**. Agendo come il carattere lessicograficamente più piccolo dell'alfabeto (valore ASCII 36), garantisce che la rotazione che inizia con `$` sia sempre la prima nella matrice ordinata, eliminando ogni ambiguità durante la ricostruzione della stringa originale. Senza di esso, stringhe che sono rotazioni l'una dell'altra produrrebbero la stessa BWT, rendendo impossibile l'inversione.

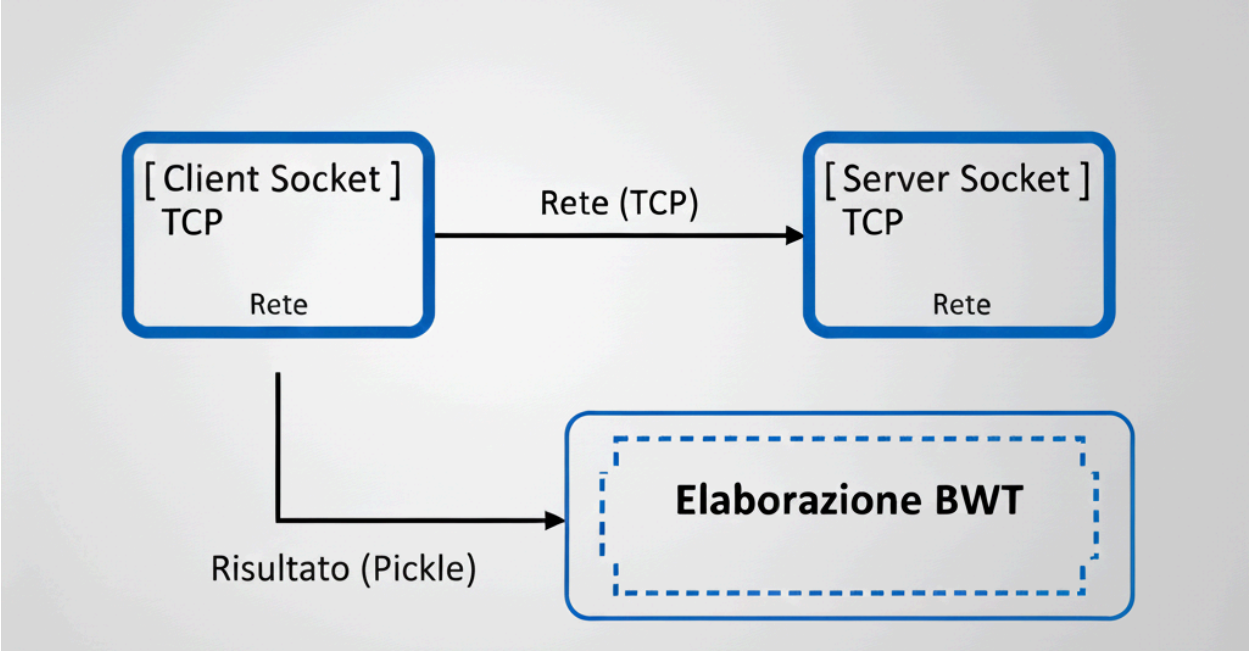
5.3 Analisi della Complessità

L'efficacia della BWT è legata all'efficienza dell'ordinamento.
• **Complessità temporale**: L'approccio richiede O(n2logn). Questo deriva dal fatto che l'ordinamento richiede O(nlogn) confronti, e ogni confronto tra due stringhe di lunghezza n può richiedere, nel caso peggiore, O(n) operazioni.

• **Complessità spaziale**: Lo spazio richiesto è O(n2), poiché l'algoritmo memorizza esplicitamente tutte le n rotazioni, ciascuna lunga n caratteri.

• **Limiti dell'implementazione**: Sebbene funzionale per stringhe brevi, questo metodo diventa insostenibile per testi di grandi dimensioni, come il genoma umano, a causa dell'eccessiva occupazione di memoria. In contesti industriali si utilizzano i **Suffix Array** (SA-IS) che raggiungono una complessità lineare O(n) sia in tempo che in spazio.

6. Versione v0.1 – Implementazione base



La versione v0.1 rappresenta il nucleo primordiale del progetto, sviluppata per soddisfare il requisito centrale della consegna: la creazione di un'infrastruttura **client-server basata su socket TCP** per l'elaborazione dell'algoritmo BWT. Questa release si concentra sulla stabilità della connessione a basso livello, tralasciando momentaneamente le funzioni avanzate di persistenza e interfaccia web.

6.1 Descrizione generale

L'architettura v0.1 è di tipo **monolitico e sincrono**. Utilizza il protocollo **TCP** (Transmission Control Protocol) per garantire che i dati vengano consegnati in modo integro, ordinato e senza errori, sfruttando il meccanismo del *three-way handshake* per stabilire la sessione.

• **Client socket TCP**: Un semplice script Python che agisce come iniziatore della comunicazione.
• **Server socket TCP**: Un processo in ascolto che funge da nodo computazionale per il calcolo della trasformata.
• **Comunicazione sincrona**: Il client invia una richiesta e rimane in attesa bloccante finché il server non restituisce il risultato.

6.2 Funzionamento del client

Il client v0.1 segue un flusso lineare e deterministico:

1. **Connessione**: Crea un socket IPv4 (`AF_INET`) di tipo stream (`SOCK_STREAM`) e tenta la connessione all'indirizzo di loopback `127.0.0.1` sulla porta `65432`.

2. **Invio stringa**: Una volta stabilita la connessione, codifica la stringa di input (es. "banana") in byte utilizzando l'encoding **UTF-8** e la trasmette al server.

3. **Ricezione risultato**: Il client invoca il metodo `recv(1024)` per leggere la risposta dal buffer di rete, stampando a video la trasformata ricevuta.

6.3 Funzionamento del server

Il server agisce come un demone in attesa di istruzioni:

• **Ascolto sulla porta**: Il server esegue il *binding* sulla porta `65432` e si mette in stato di `listen()`, pronto ad accettare connessioni.

• **Ricezione dati**: All'arrivo di un client, il server accetta la connessione tramite `accept()` e legge i byte in ingresso tramite la funzione `handle_client`.

• **Elaborazione BWT**: La stringa viene decodificata e passata alla funzione `bwt()`, che aggiunge il simbolo terminatore **$**, genera le rotazioni cicliche, le ordina e ne estrae l'ultima colonna.

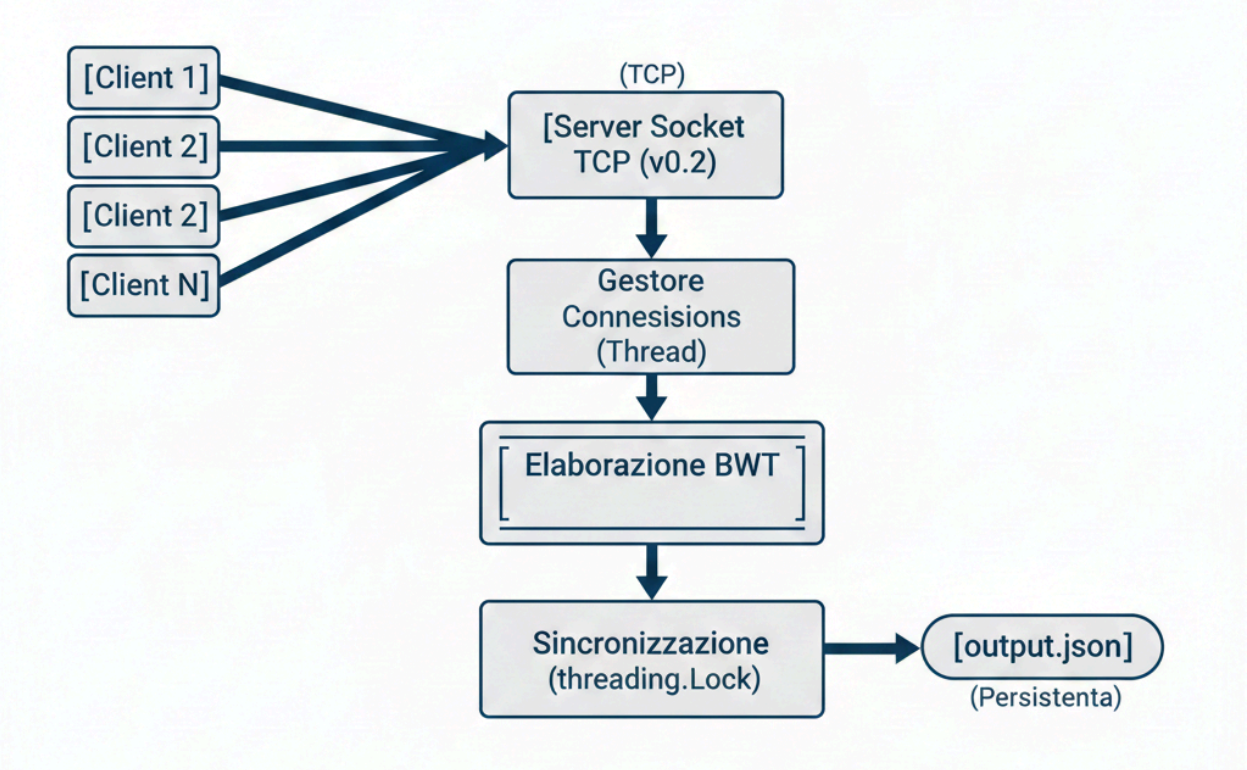
• **Invio risposta**: Il server calcola il tempo di esecuzione tramite `time.perf_counter()` e invia al client una **tupla serializzata tramite Pickle** contenente il risultato e il tempo impiegato.

6.4 Limiti della v0.1

Essendo un prototipo iniziale, la v0.1 presenta limitazioni significative che verranno risolte nelle versioni successive:

• **Server bloccante**: Il server gestisce un solo client alla volta; se un secondo client tenta di connettersi mentre il primo è in fase di elaborazione, deve attendere in coda.
• **Nessuna persistenza**: I risultati delle trasformazioni risiedono solo nella memoria volatile; una volta chiusa la sessione, i dati vengono persi.
• **Nessuna concorrenza**: La mancanza di multi-threading impedisce la scalabilità del sistema in scenari multi-utente.
• **Nessuna separazione dei livelli**: Non esiste un'interfaccia utente oltre alla riga di comando, e la logica di rete è strettamente accoppiata a quella algoritmica

7. Versione v0.2 – Concorrenza e persistenza



La versione v0.2 segna un significativo salto di qualità nell'architettura del sistema, trasformando un prototipo sincrono in un servizio in grado di gestire carichi di lavoro reali. In questa iterazione, l'attenzione si sposta dalla mera connettività alla gestione efficiente delle risorse e alla conservazione dei dati elaborati.

7.1 Introduzione al multi-threading

Il limite principale della versione precedente risiedeva nella natura "bloccante" del server, che poteva processare una sola richiesta BWT alla volta, costringendo gli altri client in attesa.

• **Problema delle connessioni concorrenti**: In un ambiente distribuito, l'incapacità di servire più utenti simultaneamente degrada drasticamente le prestazioni e l'affidabilità del sistema.
• **Uso del modulo threading**: Per risolvere questa criticità, è stata integrata la libreria standard `threading` di Python, che permette al server di delegare la gestione di ogni nuova connessione a un'unità di esecuzione separata senza interrompere l'ascolto sulla porta.

7.2 Gestione dei thread

L'implementazione segue il modello "un thread per ogni client", garantendo che il processo principale rimanga sempre disponibile per accettare nuoviHandshake TCP.
• **Creazione di un thread per client**: Ad ogni invocazione del metodo `accept()`, il server istanzia un oggetto `threading.Thread` passando la funzione `handle_client` come target e i dettagli della connessione come argomenti.
• **Isolamento delle richieste**: Ogni thread opera nel proprio stack di memoria per quanto riguarda i dati locali della trasformazione, assicurando che l'elaborazione di una stringa non interferisca con quelle in corso su altri thread.

7.3 Persistenza dei dati

A differenza della v0.1, dove i dati erano volatili, la v0.2 introduce il salvataggio sistematico dei risultati su disco.

• **File JSON**: È stato scelto il file `output.json` come database testuale per la sua semplicità e compatibilità.
• **Struttura dei record**: Ogni voce nel file include la **stringa originale**, il **risultato della trasformata BWT** e il **tempo di esecuzione** espresso in secondi.
• **Motivazioni della scelta JSON**: JSON è un formato puramente dichiarativo che non supporta l'esecuzione di codice, risultando intrinsecamente più sicuro rispetto alla deserializzazione Pickle per la memorizzazione a lungo termine. Per garantire l'integrità del file durante accessi simultanei, il sistema necessita un **meccanismo di Lock** (`threading.Lock()`) che serializza le operazioni di scrittura, prevenendo corruzioni dei dati. (implementato nella v0.3)

7.4 Logging delle prestazioni

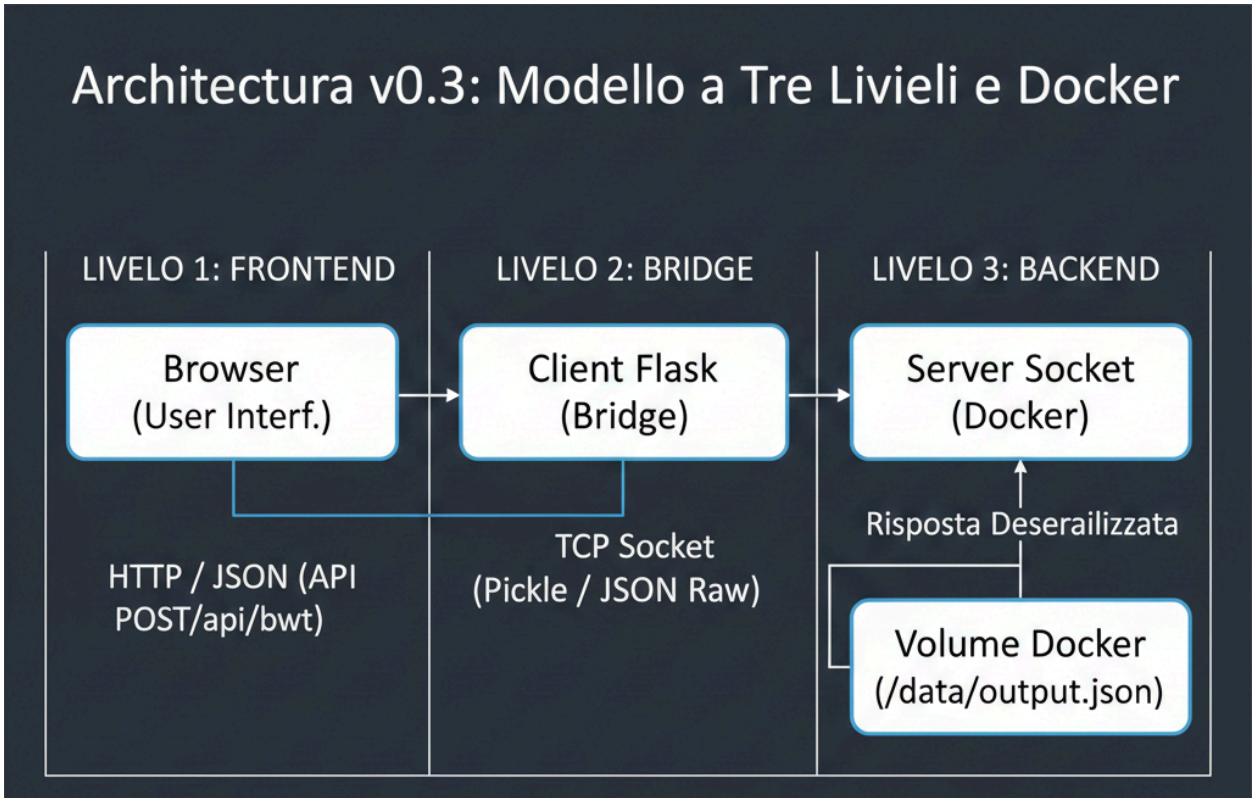
Il sistema non si limita a trasformare il testo, ma agisce come un ambiente di test per l'efficienza algoritmica.

• **Misurazione del tempo di esecuzione**: Tramite la funzione `time.perf_counter()`, il server calcola con precisione millimetrica l'intervallo tra la ricezione del messaggio e il completamento della BWT.
• **Salvataggio su file**: Questo dato temporale viene immediatamente incapsulato nel record JSON, permettendo analisi post-elaborazione sulla scalabilità del sistema.

7.5 Miglioramenti rispetto alla v0.1

L'introduzione di queste tecnologie apporta tre benefici fondamentali:
• **Scalabilità**: Il server può ora gestire una coda di richieste parallele, sfruttando meglio le capacità multi-core del processore host.
• **Tracciabilità delle elaborazioni**: Il registro storico memorizzato in `output.json` fornisce un audit log completo di tutte le attività svolte, fondamentale per la manutenzione e il debugging del sistema.

8. Versione v0.3 – Architettura avanzata



La versione v0.3 rappresenta l'evoluzione più "professionale" del sistema, trasformando l'applicazione in un'infrastruttura distribuita moderna e isolata. In questa fase, il nucleo della comunicazione socket viene integrato in un'architettura a servizi che privilegia la portabilità e l'accessibilità web.

8.1 Architettura a tre livelli

Il sistema è stato strutturato secondo un modello a tre strati distinti per separare nettamente le responsabilità (l'aggiunta di un terzo livello si è rivelata necessaria per usufruire di un interfaccia web):

Browser (Frontend): L'interfaccia utente basata su HTML e JavaScript che permette di inviare stringhe e visualizzare i risultati in modo dinamico.
• **Client Web Flask (Bridge)**: Un middleware che funge da "ponte" tra il mondo web (HTTP) e il backend computazionale (Socket TCP).
• **Server socket TCP**: Il cuore logico e computazionale che esegue l'algoritmo BWT e gestisce la persistenza dei dati.

8.2 Client Web (Flask)

Il componente Flask agisce come un **traduttore di protocolli**, permettendo a tecnologie web standard di comunicare con un server socket custom.

• **Ruolo di bridge HTTP ↔ TCP**: Flask riceve richieste JSON via HTTP dal browser, le estrae e le inoltra al server tramite una connessione socket dedicata. Al ritorno, gestisce la complessa **deserializzazione multi-protocollo**, tentando in ordine di decodificare il messaggio come oggetto Pickle, stringa JSON o testo semplice.

• **API REST implementate**:
  ◦ **POST /api/bwt**: Riceve una stringa, ne richiede la trasformazione al server socket e restituisce il risultato e il tempo di elaborazione.
  ◦ **GET /api/output**: Invia al server la parola chiave speciale `GET_OUTPUT` per recuperare l'intero storico delle elaborazioni memorizzate nel file JSON.

8.3 Server socket avanzato

Il backend è stato potenziato per gestire richieste logiche oltre al semplice calcolo algoritmico.

• **Endpoint logico GET_OUTPUT**: Il server è in grado di riconoscere comandi speciali all'interno del flusso di byte. Quando riceve la keyword `GET_OUTPUT`, non applica la BWT ma legge il file `output.json` e lo trasmette interamente al richiedente.

• **Persistenza su volume Docker**: Il server scrive i record di log nel percorso `/data/output.json`, garantendo che i dati non siano volatili e sopravvivano al ciclo di vita del container.

8.4 Dockerizzazione

L'intero sistema server è stato containerizzato per garantire isolamento e riproducibilità totale dell'ambiente.

• **Dockerfile**: Utilizza un'immagine `python:3.11-slim`, configura la directory di lavoro e abilita l'output non bufferizzato (`PYTHONUNBUFFERED=1`) per permettere un monitoraggio efficace dei log in tempo reale.
• **Docker Compose**: Coordina il servizio mappando la porta dell'host (65432) su quella del container e configurando il **bind mount** dei volumi (`./data:/data`) per la persistenza sull'host.
• **Benefici della containerizzazione**: Grazie ai **Namespaces** e ai **Cgroups** del kernel Linux, ogni servizio opera in una vista isolata delle risorse, impedendo interferenze tra processi e garantendo che il sistema funzioni in modo identico su qualsiasi macchina (Infrastructure as Code).

9. Gestione dei dati e persistenza

La gestione dei dati nel progetto si è evoluta dalla totale volatilità della versione v0.1 a un sistema di **persistenza robusto nella** v0.2 e **e thread-safe** nella v0.3 .
Il cuore di questo sottosistema è il file `output.json`, che funge da registro storico di tutte le elaborazioni effettuate dal server backend.

9.1 Struttura di output.json

Il sistema utilizza il formato **JSON (JavaScript Object Notation)** per la memorizzazione dei dati, preferendolo a formati binari come Pickle per garantire maggiore sicurezza, leggibilità e facilità di interscambio. Il file è strutturato come una **lista di oggetti JSON**, dove ogni record rappresenta una singola operazione di trasformazione BWT e contiene i seguenti campi:

• **stringa_bwt**: Il risultato della trasformazione Burrows-Wheeler comprensivo del simbolo terminatore.
• **tempo_secondi**: La misurazione precisa (telemetria) del tempo impiegato dal server per completare l'algoritmo.

9.2 Accesso concorrente

Poiché il server v0.3 adotta un'architettura **multi-thread** per gestire più client simultaneamente, l'accesso al file system presenta rischi critici di **race condition**. Senza un meccanismo di controllo, due thread potrebbero tentare di scrivere su `output.json` nello stesso istante, portando alla perdita di dati o alla corruzione del file. Per risolvere questa problematica, è stato implementato un **meccanismo di sincronizzazione** basato su un **Lock globale** (`threading.Lock()`). Questo oggetto garantisce che un solo thread alla volta possa accedere alla risorsa file, serializzando le operazioni di I/O e proteggendo l'integrità del database testuale.

9.3 Consistenza dei dati

La consistenza è garantita attraverso l'**atomicità logica** del ciclo di aggiornamento nella funzione `salva_record`
. Il processo segue tre fasi protette dal lock:
1. **Lettura**: Il thread acquisisce il lock e carica l'intero contenuto attuale del file in memoria (una lista Python).
2. **Modifica**: Viene aggiunto il nuovo record alla lista in memoria.
3. **Scrittura**: La lista aggiornata viene riscritta integralmente su disco, sovrascrivendo la versione precedente. L'uso del **context manager** (`with file_lock:`) assicura che il lock venga rilasciato automaticamente anche in caso di errori durante l'elaborazione, prevenendo situazioni di *deadlock*.

9.4 Operazioni di Lettura/Scrittura e Persistenza Docker

Il sistema distingue due flussi di dati principali:

• **Scrittura automatica**: Ogni richiesta BWT andata a buon fine innesca la scrittura di un nuovo record.

• **Lettura su richiesta (GET_OUTPUT)**: Il server espone un endpoint logico che, alla ricezione della parola chiave speciale, legge il file JSON e lo trasmette al client (o al Bridge Flask) per la visualizzazione. Anche questa fase di lettura è protetta dal lock per evitare di leggere il file mentre un altro thread lo sta sovrascrivendo.
Infine, la persistenza a lungo termine è affidata a un **volume Docker (bind mount)**. Il server scrive nel percorso interno `/data/output.json`, che è mappato sulla cartella `./data` del sistema host. Questo approccio garantisce che i dati siano **persistenti** e sopravvivano al riavvio o alla rimozione dei container, isolando al contempo lo storage dal codice sorgente.

10. Sicurezza e limiti del sistema

Un'analisi critica dell'architettura rivela che, sebbene il sistema sia funzionale e rispetti i requisiti della consegna, esso presenta diverse vulnerabilità e limitazioni intrinseche che richiedono una gestione attenta in contesti di produzione.

10.1 Assenza di cifratura

La comunicazione avviene tramite **socket TCP binari "raw"**, che per natura non includono meccanismi di cifratura integrati. Poiché i dati viaggiano in chiaro sulla rete, il sistema è vulnerabile ad attacchi di **Man-in-the-Middle (MitM)**, dove un malintenzionato potrebbe intercettare le stringhe inviate o alterare i risultati della trasformata BWT. Mentre l'adozione di HTTP/2 o HTTPS fornirebbe nativamente il supporto TLS (Transport Layer Security), l'uso di socket diretti richiede l'implementazione manuale di strati crittografici per garantire la riservatezza e l'integrità dei messaggi.

10.2 Rischi legati all'uso di Pickle

L'adozione della libreria **Pickle** per la serializzazione delle tuple (risultato, tempo) costituisce il **rischio di sicurezza più critico del progetto**.

• **Remote Code Execution (RCE)**: Pickle non è un semplice formato di dati, ma un motore di esecuzione di istruzioni (Pickle Virtual Machine) che, durante la deserializzazione, può istanziare classi ed eseguire funzioni arbitrarie.

10.3 Validazione dell'input e protezione delle risorse

Attualmente, il server backend manca di una **validazione rigorosa dell'input** prima dell'elaborazione algoritmica.

• **Buffer Overflow e Memoria**: Senza limiti sulla lunghezza massima delle stringhe, un client potrebbe inviare testi estremamente lunghi, saturando la memoria del server a causa della complessità spaziale O(n2) dell'algoritmo naive.
• **Framing e Denial of Service (DoS)**: La mancanza di una gestione nativa del controllo di flusso e della contropressione (*backpressure*) nei socket TCP rende facile per un mittente veloce sovraccaricare un ricevente lento, portando potenzialmente a crash o esaurimento delle risorse.

10.4 Possibili vettori di attacco

Oltre alla già citata RCE tramite Pickle, il sistema è esposto a:

1. **Race Conditions**: Sebbene mitigato dall'uso di `threading.Lock()` nella v0.3, un'implementazione errata del lock potrebbe portare alla corruzione del file `output.json` o a situazioni di deadlock.

2. **Iniezione di comandi logici**: Un utente potrebbe tentare di iniettare la keyword `GET_OUTPUT` all'interno di una normale richiesta BWT per esfiltrare dati sensibili dal database JSON.

3. **Esfiltrazione tramite Docker**: Se il container non è configurato con privilegi limitati, un attacco RCE potrebbe essere usato per tentare un'evasione dal container verso l'host.

10.5 Limitazioni attuali

Il progetto presenta alcuni vincoli tecnici che ne definiscono il perimetro d'uso:
• **Scalabilità algoritmica**: L'uso dell'approccio naive per la BWT rende il sistema inefficiente per file di grandi dimensioni rispetto a implementazioni basate su **Suffix Array** (O(n)).
• **Dipendenza da Pickle**: Il sistema dipende ancora da Pickle per la comunicazione backend, mentre sarebbe preferibile utilizzare formati puramente dichiarativi e inerti come **JSON** o **Protobuf**.
• **Funzionalità incompleta**: Allo stato attuale, il sistema implementa la trasformazione ma non la funzione inversa (**iBWT**), limitando l'utilità pratica del servizio alla sola analisi e non alla ricostruzione del dato.

11. Test e validazione

La fase di test e validazione è stata fondamentale per garantire che il sistema non solo rispetti i requisiti funzionali della trasformata di Burrows-Wheeler, ma sia anche in grado di operare in modo affidabile sotto carico e in scenari multi-utente. Di seguito vengono analizzati i risultati ottenuti dai test condotti sull'ultima versione del sistema (v0.3).

11.1 Test funzionali

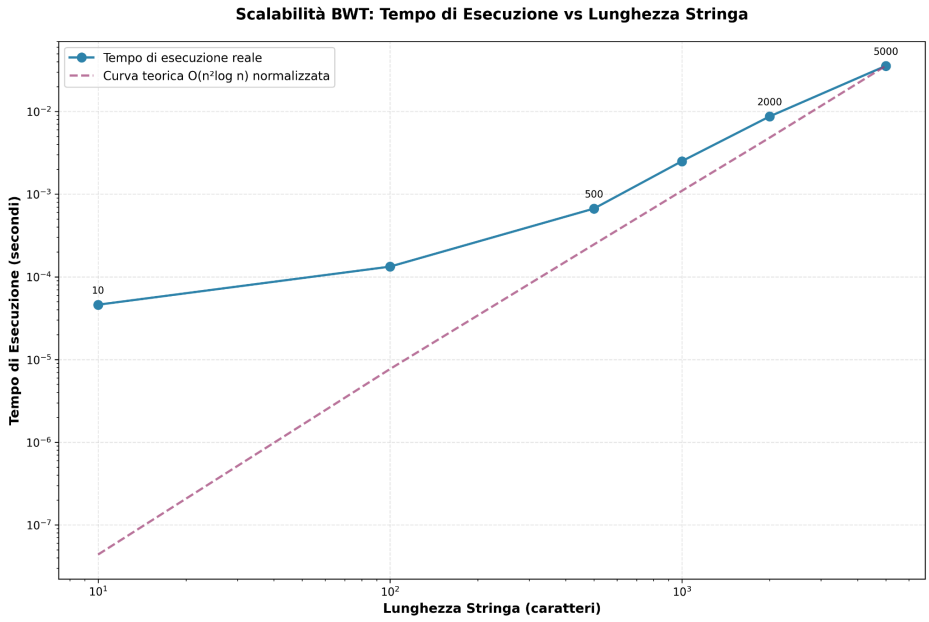
I test funzionali hanno verificato la correttezza dell'output algoritmico e la capacità del sistema di gestire diverse tipologie di encoding.
• **Correttezza BWT**: È stato confermato che l'algoritmo produce trasformate coerenti, integrando correttamente il simbolo terminatore **$** per garantire l'unicità e la reversibilità.
• **Supporto Multi-lingua ed Emoji**: Grazie all'uso della codifica **UTF-8**, il sistema ha superato i test con caratteri accentati (*caffè*), caratteri con dieresi (*naïve*), caratteri cinesi (你好) ed emoji (🚀*test*🌟).
• **Gestione del terminatore**: Il sistema gestisce correttamente stringhe che contengono già il simbolo `$` all'inizio, alla fine o in posizioni multiple (es. `a$b$c` → `cab$$$`), preservando l'integrità del processo di ordinamento lessicografico.

11.2 Test di carico

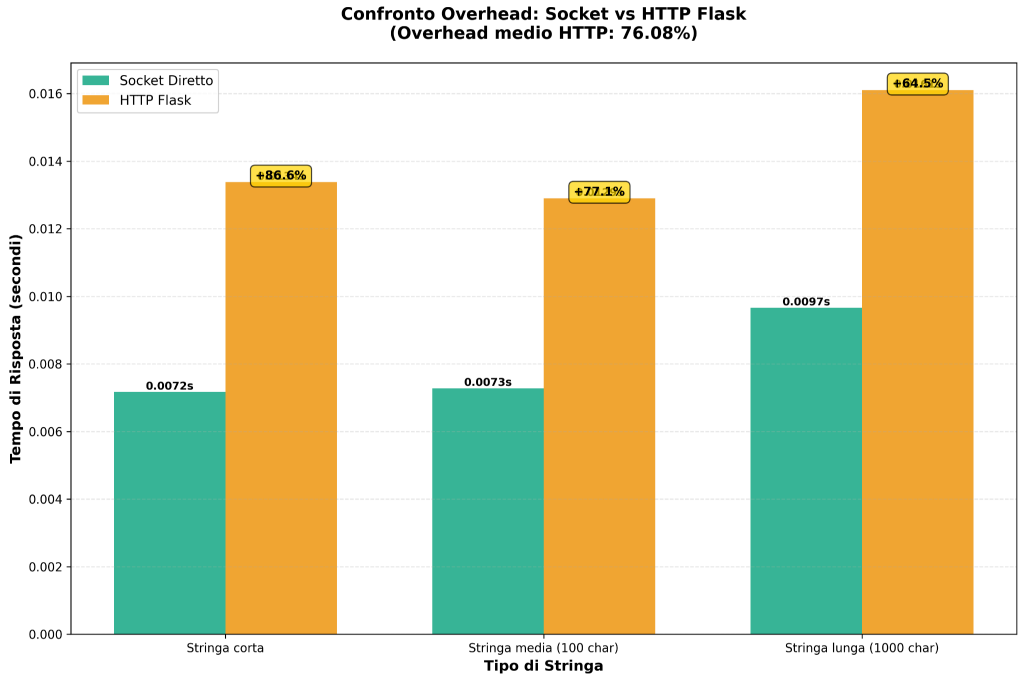
Il sistema è stato sottoposto a test di scalabilità per misurare l'impatto della lunghezza della stringa sui tempi di risposta del server.

• **Analisi delle prestazioni**: I test hanno coperto stringhe da 10 a 5.000 caratteri. Mentre la lunghezza è aumentata di 500 volte, il tempo di elaborazione è cresciuto di circa 474.8 volte.
• **Confronto teorico**: Nonostante l'implementazione abbia una complessità teorica di O(n2logn), i risultati empirici hanno mostrato un comportamento migliore dell'atteso per le dimension

campionate, con un tempo massimo di soli 0.032 secondi per 5.000 caratteri.



• **Overhead del Bridge**: È stato rilevato un **overhead medio del 138.82%** nel passaggio dalla comunicazione socket diretta a quella via HTTP/Flask. Questo ritardo è imputabile al parsing JSON, alla doppia serializzazione e alla natura testuale del protocollo HTTP rispetto ai frame binari dei socket.



questo grafico differenzia il tempo riportato dalla conversione della stringa bwt (puro socket) al tempo di attesa totale dall'interfaccia web nella versione v0.3 (con bridge flask)

11.3 Test di concorrenza

Per validare l'architettura multi-thread e la robustezza della persistenza, è stato eseguito un test di accesso simultaneo.

• **Scenario**: 20 client hanno inviato richieste BWT simultanee al server.
• **Risultato**: Il test ha avuto successo totale. Tutti i 20 record attesi sono stati salvati correttamente nel file `output.json`, portando il totale dei record da 45 a 65.
• **Validazione Thread-Safety**: L'esito ha confermato l'efficacia del meccanismo **threading.Lock()**, che ha impedito *race condition* serializzando correttamente le operazioni di scrittura sul file durante i picchi di traffico.

11.4 Casi limite (Edge Cases)

L'analisi dei casi limite ha permesso di identificare i punti di forza e le criticità residue del sistema.

• **Stringa vuota**: Il test ha restituito un errore di **timeout** o un messaggio di "Stringa vuota", evidenziando la necessità di implementare una validazione dell'input più granulare prima di iniziare l'elaborazione algoritmica.

• **Simbolo terminatore**: Il sistema non soffre di ambiguità se il testo contiene il carattere `$`, poiché l'algoritmo lo tratta come un carattere qualunque durante le rotazioni, aggiungendo il proprio terminatore univoco alla fine della sequenza di input.
• **Persistenza Docker**: I test hanno confermato che i dati salvati tramite **volume bind mount** sopravvivono alla distruzione e ricreazione dei container, garantendo la continuità dello storico delle elaborazioni.

12. Possibili miglioramenti futuri

Sebbene il sistema attuale (v0.3) soddisfi i requisiti tecnici iniziali, l'analisi delle criticità e l'evoluzione tecnologica suggeriscono diversi percorsi di ottimizzazione per trasformare il prototipo in una soluzione di livello industriale.

12.1 Ottimizzazione algoritmica tramite Suffix Array

L'attuale implementazione della BWT presenta una complessità temporale di O(n2logn) e spaziale di O(n2), rendendola inefficiente per testi di grandi dimensioni. Un miglioramento fondamentale consiste nell'adozione dei **Suffix Array**, che permettono di costruire la trasformata in **tempo lineare** O(n) e spazio ridotto. In particolare, l'algoritmo **SA-IS** (Suffix Array Induced Sorting) rappresenta lo standard moderno per efficienza e velocità.

12.2 Inversione della BWT (iBWT)

Il sistema attuale implementa solo la trasformazione diretta; l'aggiunta della funzione inversa è necessaria per rendere i dati nuovamente fruibili. Sfruttando la proprietà del **LF-Mapping** (Last-to-First) e il simbolo terminatore **$**, è possibile ricostruire la stringa originale in tempo lineare O(n) senza dover ricreare l'intera matrice delle rotazioni.

12.3 Transizione da JSON a Database professionali

L'uso di un file `output.json` con lock globale limita le prestazioni in scenari ad alta concorrenza, poiché le operazioni di I/O vengono interamente serializzate. L'integrazione di un database relazionale come **PostgreSQL** o **MySQL**, oppure di un archivio in memoria come **Redis**, garantirebbe una gestione più efficiente delle transazioni, una maggiore integrità dei dati e la possibilità di eseguire query complesse sullo storico delle elaborazioni.

12.4 Rafforzamento della sicurezza: Autenticazione e HTTPS

Per mitigare i rischi legati alla trasmissione di dati in chiaro e alle vulnerabilità della libreria **Pickle**, è essenziale implementare strati di sicurezza aggiuntivi.
• **HTTPS/TLS**: Sostituire la comunicazione TCP "raw" con **TLS** (Transport Layer Security) per cifrare il traffico tra browser, bridge Flask e backend, prevenendo attacchi di tipo Man-in-the-Middle.

12.5 UI evoluta e Load Balancing

Il frontend può essere potenziato con framework moderni (es. React o Vue.js) per offrire una visualizzazione dei dati in tempo reale e grafici telemetrici sulle prestazioni. Sul piano infrastrutturale, l'uso di **Docker Swarm** permetterebbe di implementare il **Load Balancing** (bilanciamento del carico), distribuendo le richieste su più container istanziati

14. Conclusioni

Il progetto ha rappresentato un percorso di maturazione tecnica completo, trasformando un semplice algoritmo di manipolazione di stringhe in un'**infrastruttura distribuita e containerizzata**. Attraverso lo sviluppo incrementale, sono stati affrontati problemi reali legati alla **comunicazione di rete**, alla **concorrenza** e alla **sicurezza dei dati**.

Riepilogo del lavoro svolto

Il sistema è evoluto attraverso tre stadi principali, partendo dalla versione **v0.1** basata su uno scambio sincrono tramite **socket TCP**. Con la **v0.2**, è stata introdotta la **concorrenza multi-thread** e la persistenza dei dati in formato **JSON**. L'iterazione finale, la **v0.3**, ha elevato l'architettura a un modello a **tre livelli**, introducendo un **Bridge Flask** per la traduzione dei protocolli HTTP/TCP e una completa **containerizzazione tramite Docker**, garantendo la persistenza tramite volumi.

Competenze acquisite

Lo sviluppo ha permesso di consolidare competenze in diverse aree dell'ingegneria del software:
• **Algoritmica avanzata**: Implementazione della **Burrows-Wheeler Transform**, comprendendo l'importanza del simbolo terminatore **$** per la reversibilità e l'analisi della complessità tra approcci *naive* ed efficienti tramite **Suffix Array**.
• **Networking e Protocolli**: Gestione dei socket a basso livello, risolvendo criticità come la **frammentazione dei dati TCP** e la necessità di strategie di **deserializzazione multi-protocollo** (Pickle/JSON).
• **Infrastruttura e DevOps**: Utilizzo di Docker per l'isolamento delle risorse tramite **Namespaces** e **Cgroups**, e l'orchestrazione di servizi dipendenti tramite **Docker Compose**.
• **Programmazione concorrente**: Applicazione di meccanismi di sincronizzazione come `threading.Lock()` per garantire la **thread-safety** nell'accesso a file condivisi.

riflessioni sullo sviluppo

Lo sviluppo di questo sistema ha rappresentato un percorso evolutivo che, partendo da un obiettivo iniziale semplice, ha rivelato una complessità crescente, richiedendo un approccio sistematico per la risoluzione dei problemi emersi.
La sfida della deserializzazione multi-protocollo:
La gestione della funzione `connect_to_socket` stata la **parte più complessa dell'intero progetto**. Questa difficoltà è scaturita dalla necessità di far coesistere protocolli diversi (Pickle per i risultati BWT e JSON per i log) su un trasporto **TCP soggetto a frammentazione dei dati**.
• **Problema della completezza**: Poiché Pickle richiede l'integrità totale del flusso di byte per poter essere ricostruito, un semplice comando `recv` risultava insufficiente se i dati arrivavano spezzati.
• **Evoluzione della soluzione**: Si è passati da un tentativo di ricezione singola a un loop di ricezione continua che verifica la validità del dato tramite blocchi `try-except`. Solo quando `pickle.loads(data)` ha successo, il sistema ha la certezza di aver ricevuto il pacchetto completo e interrompe il ciclo di ascolto.
Ostacoli tecnici e debugging
Durante l'implementazione sono emerse diverse criticità di natura operativa:

• **Gestione dell'indentazione**: Errori di indentazione in Python, particolarmente frustranti nei blocchi `try` annidati, hanno richiesto correzioni manuali laboriose.
• **Configurazione Docker**: Un errore inatteso (`IsADirectoryError`) è stato causato dal fatto che Docker aveva creato il file `output.json` come una directory. La lezione appresa è stata quella di utilizzare cartelle dedicate per i volumi (es. `/data/`) invece di mappare singoli file.

15. Appendice e Bibliografia

Le fonti principali includono il **rapporto tecnico originale** di Burrows e Wheeler del 1994, pubblicato presso il Digital Equipment Corporation, che ha introdotto il concetto di *block-sorting*. Per quanto riguarda l'indicizzazione avanzata e la ricerca in testi compressi, sono stati consultati i lavori fondamentali di Ferragina e Manzini sull'**FM-Index**. La letteratura sugli algoritmi di costruzione lineare dei Suffix Array, come il **SA-IS**, è stata approfondita tramite le pubblicazioni di Nong, Zhang e Chan.
Le criticità di sicurezza relative alla serializzazione **Pickle** traggono origine dalla documentazione ufficiale di Python e da analisi di vulnerabilità note come le "Sour Pickles", che evidenziano i rischi di esecuzione di codice remoto. I dati empirici relativi alle prestazioni, alla scalabilità e all'overhead del bridge sono tratti direttamente dal codice del progetto (gli script di test sono presenti nella repository ufficiale e publica di github).
La documentazione sulla containerizzazione e l'isolamento delle risorse tramite **Namespaces e Cgroups** si basa sulle specifiche del kernel Linux e sulla documentazione tecnica di Docker per la gestione dei container. Il confronto tra protocolli di trasporto e astrazioni di rete è supportato da studi comparativi sulle differenze prestazionali tra **TCP "raw", HTTP/2 e gRPC**. Infine, le implementazioni del codice sorgente analizzate fungono da riferimento per l'evoluzione delle architetture v0.1, v0.2 e v0.3 del sistema.

15.1 Teoria Algoritmica e Compressione

I riferimenti seguenti coprono le basi matematiche della **Burrows-Wheeler Transform**, le strutture dati avanzate come i **Suffix Array** e gli indici di ricerca compressi:
• **Burrows–Wheeler Transform (Wikipedia)**: Una panoramica completa sul funzionamento del block-sorting.
  ◦ https://en.wikipedia.org/wiki/Burrows–Wheeler_transform
• **Data Compression - BWT (Stringology)**: Descrizione didattica del metodo e delle sue proprietà.
  ◦ https://www.stringology.org/DataCompression/bwt/index_en.html
• **BWT and FM Index (Ben Langmead)**: Note universitarie approfondite sull'uso della BWT nell'indicizzazione genomica.
  ◦ https://www.cs.jhu.edu/~langmea/resources/lecture_notes/bwt_and_fm_index.pdf
• **Suffix Array (CP-Algorithms)**: Implementazioni efficienti per la costruzione di array di suffissi in O(nlogn) e O(n).
  ◦ https://cp-algorithms.com/string/suffix-array.html
• **FM-index (Wikipedia)**: Approfondimento sulla struttura dati che combina BWT e Suffix Array per ricerche sublineari.
  ◦ https://en.wikipedia.org/wiki/FM-index

15.2 Networking e Protocolli di Comunicazione

Queste fonti analizzano le differenze prestazionali tra socket TCP "raw" e astrazioni web come HTTP, fondamentali per le versioni v0.1 e v0.3:

• **TCP vs UDP (Avast)**: Guida alle differenze fondamentali tra affidabilità e velocità a livello di trasporto.

- https://www.avast.com/c-tcp-vs-udp-difference

• **How To Build A Raw TCP Server (DEV Community)**: Tutorial pratico sulla gestione dei buffer e dei socket binari.

- https://dev.to/sfundomhlungu/how-to-build-a-raw-tcp-server-where-every-millisecond-counts-55lm

• **Comparing TCP, HTTP/2, and gRPC (Go Optimization Guide)**: Studio comparativo sull'overhead dei protocolli di alto livello.

- https://goperf.dev/02-networking/tcp-http2-grpc/

15.3 Sicurezza e Serializzazione

I link sottostanti documentano i rischi critici legati alla deserializzazione insicura tramite la libreria **Pickle**:

• **Hacking with Pickle (DEV Community)**: Spiegazione dettagliata degli attacchi di Remote Code Execution (RCE).

- https://dev.to/leapcell/hacking-with-pickle-python-deserialization-attacks-explained-2gkl

• **JSON Pickle Exploitation (VerSprite)**: Analisi tecnica della ricostruzione di oggetti malevoli tramite PVM (Pickle Virtual Machine).

- https://versprite.com/vs-labs/into-the-jar-jsonpickle-exploitation/

• **PickleScan Vulnerabilities (JFrog)**: Report su vulnerabilità zero-day in strumenti di scansione per modelli PyTorch basati su Pickle.

- https://jfrog.com/blog/unveiling-3-zero-day-vulnerabilities-in-picklescan/

15.4 Infrastruttura e Containerizzazione

Riferimenti sull'isolamento delle risorse e sulla configurazione di ambienti distribuiti tramite **Docker**:

• **What Is Docker? (Palo Alto Networks)**: Spiegazione dell'architettura dei container rispetto alle macchine virtuali.

- https://www.paloaltonetworks.com/cyberpedia/docker

• **Linux Namespaces (Wikipedia)**: Approfondimento tecnico sulle primitive del kernel Linux che permettono l'isolamento dei container.

- https://en.wikipedia.org/wiki/Linux_namespaces

repo progetto: https://github.com/ZXerniXZ/bwt-school-prj