Relazione 1° Homework

Sistemi distribuiti

Gruppo:

- *Cristiano Pistorio 1000067656*
- *Giuseppe Romano 1000056390*

Sommario

Abstract 2

Diagramma architetturale 3

gRPC 3

Server 4

Diagrammi di sequenza 6

DataCollector e CircuitBreaker 6

Diagramma delle sequenze 7

DataBase 8

Container 8

Realizzazione del DockerFile 8

Creazione delle Immagini Docker 9

Configurazione della Rete tra Container 9

Client gRPC 9

Abstract

Il progetto prevede lo sviluppo di un sistema distribuito progettato per gestire informazioni finanziarie, con un focus su robustezza, scalabilità e affidabilità. La soluzione è strutturata su un'architettura a microservizi, con un server gRPC per l'interazione fra il client e il server. Il server supporta operazioni di registrazione, aggiornamento e cancellazione utenti, implementando attraverso una cache politiche di gestione dei dati con garanzia "at-most-once" per prevenire duplicazioni di richieste.

Un componente DataCollector recupera ogni minuto informazioni finanziarie utilizzando la libreria yfinance, raccogliendo i dati per gli utenti registrati. Questi dati sono protetti da un Circuit Breaker per assicurare resilienza in caso di errori o ritardi. Il database MySQL associato memorizza sia i dati utente che le informazioni finanziarie, quindi il server gRPC e il DataCollector comunicano attraverso il DB.

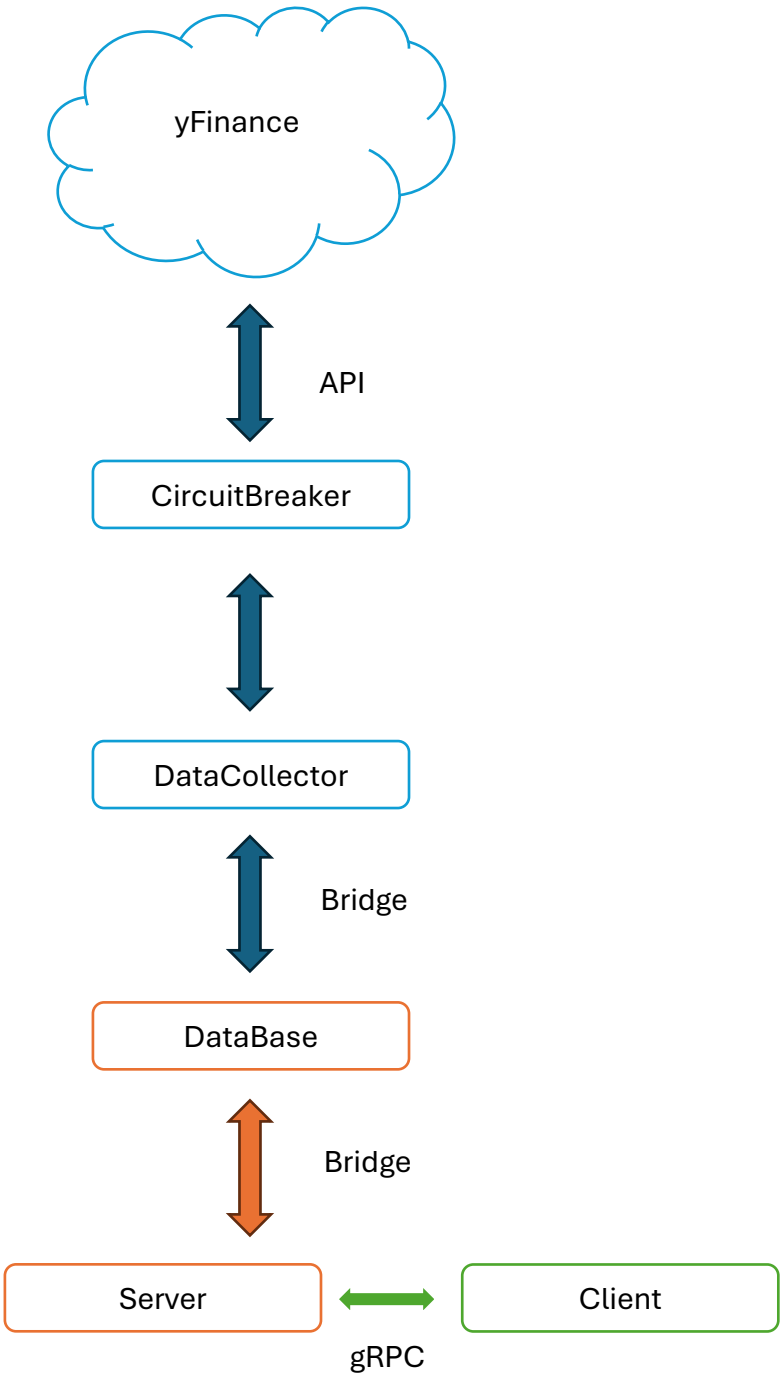
Assumiamo che ogni utente sia riconosciuto attraverso l'email. Ogni utente potrà tenere traccia di un'unica azione.

L'intero sistema è containerizzato tramite Docker Compose.

È stato implementato anche un client, cui interfaccia è un semplice menù da terminale. Le operazioni da qui eseguibili sono:

- Registra utente
- Elimina utente
- Aggiorna utente
- Mostra tutti i dati
- Mostra ultimo valore azionario (ultimo valore della azione a cui dei registrato)
- Calcola il valore medio delle ultime X azioni

Diagramma architetturale



gRPC

Il framework gRPC è il meccanismo che regola la comunicazione fra client-server. In questo modo un'applicazione client può chiamare un applicazione server come se il servizio fosse sulla stessa macchina.

La prima cosa fatta è stata delineare i servizi da fornire al client.

Uno volta stabiliti questi è stato possibile realizzare il file *user.proto*.

Ora è possibile eseguire il comando:

```
python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. user.proto
```

Questo comando (assicurandosi di avere installato gRPC nel proprio enviroment) genera i file: *user_pb2* e *user_pb2_grpc* che permettono al client e server di dialogare in modo trasparente.

Server

Il compito del server è quello di gestire gli utenti. Le richieste arriveranno sempre dai clients attraverso gRPC.

Tutto il codice necessario al funzionamento del sever è presente nel file server.py

Le funzioni che deve compiere sono:

- **Registrazione degli utenti**
  La registrazione dell'utente avviene attraverso una semplice query.
  Prima di accedere al database è opportuno verificare se la richiesta è stata già fatta, al fin di ottenere questo è stata implementata un cache che permette di gestire in modo corretto la politica "at-most-once".
  Questa politica è un principio utilizzato per garantire che una specifica azione o operazione venga eseguita non più di una volta. Questa politica evita duplicazioni o ridondanze, assicurando che ogni richiesta sia elaborata una sola volta anche se viene ritentata a causa di errori di comunicazione o altri problemi.

  A livello implementativo abbiamo deciso di realizzare la cache attraverso l'uso di tre dizionari: uno per le operazioni di registrazione, uno per le operazioni di eliminazione e uno per le operazioni di aggiornamento.
  Nello specifico nel caso dei dizionari per le operazioni di registrazione ed eliminazione la chiave è l'e-mail dell'utente, mentre il valore rappresenta lo stato: '0' operazione in corso, '1' operazione terminata in modo corretto. Se l'operazione non è nel dizionario vorrà dire che non è in cache e deve essere completata dialogando col DB.
  Questa implementazione della cache ha reso necessario affrontare un problema relativo al seguente flusso di operazioni: registrazione dell'utente X, eliminazione dell'utente X, nuova registrazione dell'utente X. Se questo problema non venisse gestito in alcun modo la prima registrazione sarebbe tenuta in cache e renderebbe impossibile ulteriori registrazioni, per evitare ciò

durante l’eliminazione dobbiamo attenzionare anche la cache avendo cura di eliminare eventuali richieste, del determinato utente, dal dizionario relativo alle registrazioni.

La registrazione gestisce tutte le eventuali eccezioni generate dal DB.

- **Elimina utente**
  La funzione è esattamente duale alla registrazione.

- **Aggiorna utente**
  L’aggiornamento dell’utente avviene attraverso una semplice query.
  Anche in questo caso prima di accedere al database è opportuno verificare se la richiesta è stata già fatta, attraverso la cache che permette di gestire la politica “at-most-once”.
  In questo caso, il dizionario non può utilizzare solo l'email come chiave, poiché ciò impedirebbe l'esecuzione di più aggiornamenti relativi alla stessa email. Questo accadrebbe perché il sistema non riconoscerebbe una nuova richiesta come distinta dalle precedenti. Utilizzando una coppia di chiavi composta dall'email normalizzata e dal ticker, possiamo gestire correttamente le richieste multiple per lo stesso utente, garantendo che ogni aggiornamento sia trattato come un'operazione unica e indipendente.

- **Mostra tutti i dati**
  Questa tool è stato implementato principalmente per vedere lo stato del sistema e se tutto sta funzionando in modo corretto. Il funzionamento è molto semplice. Si esegue la query e si mostra il risultato, non servono un gestione estremamente rigorosa delle eccezioni perché è un funzione destinata a chi vuole testare l’app.

- **Mostra ultimo valore azionario (ultimo valore della azione a cui dei registrato)**
  Questa funzione non modifica lo stato del sistema, non ha senso applicare la politica at-most-once. Si esegue semplicemente la query e si mostra il risultato.
  Una azione necessaria al funzionamento del db è che teniamo venga attenzionata è la seguente:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED
```

  Attraverso questa operazione garantiamo che tutte le operazioni di lettura in quella sessione vedranno le modifiche già commesse da altre transazioni. Questo livello di isolamento assicura che i dati letti siano consistenti, senza

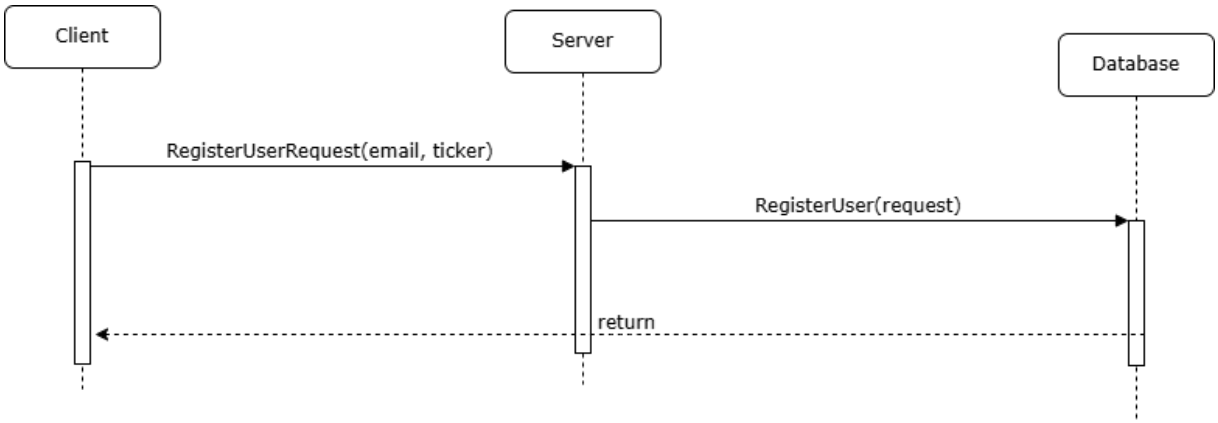
questo comando SQL abbiamo notato che i dati non venivano letti correttamente, ma venivano letti i dati di un snapshot risalente all'ultimo commit.

- **Calcola il valore medio delle ultime X azioni**
  Logicamente funziona in modo del tutto paragonabile alla funzione che mostra l'ultimo valore. La differenza è che vengono estratti gli ultimi X valori, specificati dal client in fase di richiesta, e si fa la media aritmetica.

- **Eliminazione dei vecchi dati**
  Questa funzione è stata realizzate per evitare di appesantire eccessivamente il DB. Una volta invocata l'utente può inserire un periodo di tempo definito in secondi, la funzione eliminerà tutti i dati raccolti prima del periodo di tempo inserito.

Diagrammi di sequenza

Quello che segue è il diagramma di sequenza dell' operazione di registrazione tutti gli altri diagrammi delle operazioni svolte da server sono analoghi



DataCollector e CircuitBreaker

Il DataColletor è il servizio che raccoglie i dati nella nostra applicazione. Il suo compito è quello di controllare quali sono gli utenti registrati e con quali ticker. Quindi le azioni che il DataCollector esegue periodicamente sono:

- Il recupero degli utenti attualmente registrati e i rispettivi ticker in modo da poter creare un dizionario di ticker.
- Verrà usato tale dizionario di ticker per recupere il valore delle azioni di tutti i ticker che hanno almeno un utente associato tali valori verranno recuperati da yfinance

- I valori recuperati saranno inseriti in una determinata tabella chiamata stock_prices dove ogni riga è composta da ticker, valore dell'azione e timestamp dell'operazione.
- I valori vengono inseriti ogni 60 secondi, pensiamo possa essere un buon periodo per non appesantire troppo il DB ma allo stesso tempo avere sempre un valore azionario abbastanza recente

Inoltre, per garantire robustezza del sistema si è usato un CircuitBreaker in tutte le operazioni che fanno riferimento a yfinance , questo componente è essenziale per gestire fallimenti temporanei e prevenire il sovraccarico del sistema.

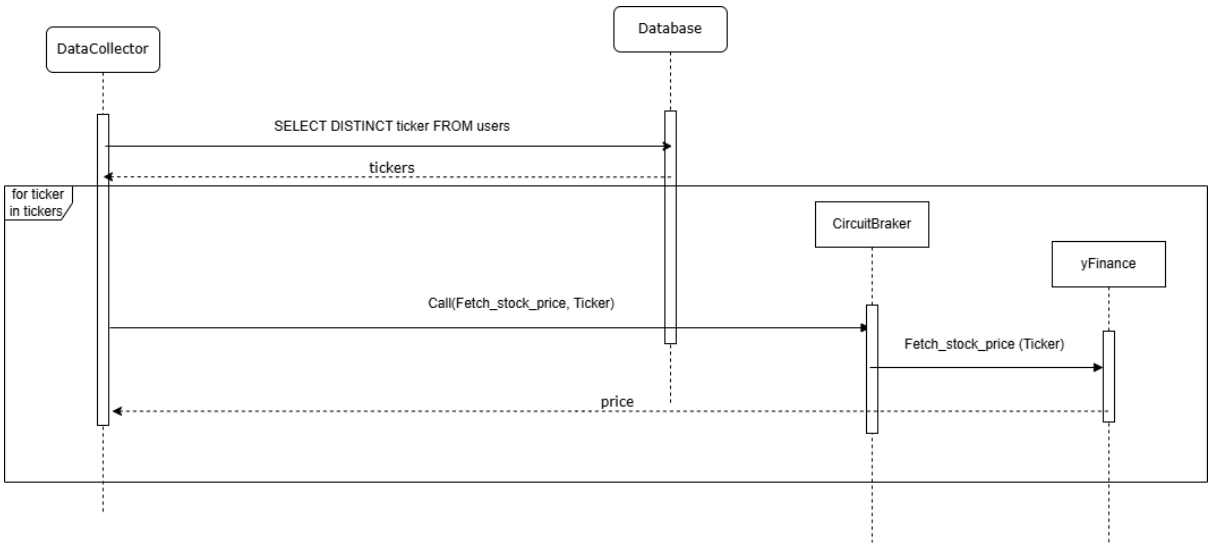
l CircuitBreaker funziona come segue:

1. **Soglia di Fallimento**: Se il numero di fallimenti consecutivi supera una soglia predefinita (failure_threshold), il CircuitBreaker entra nello stato "OPEN".
2. **Timeout di Recupero**: Dopo un periodo di timeout (recovery_timeout), il CircuitBreaker passa allo stato "HALF-OPEN" e tenta di eseguire nuovamente le operazioni.
3. **Reset dei Fallimenti**: Se l'operazione riesce durante lo stato "HALF-OPEN", il CircuitBreaker si resetta e ritorna allo stato "CLOSED". In caso contrario, rimane nello stato "OPEN" fino al prossimo tentativo.

Questo approccio permette di gestire in modo efficace i problemi temporanei, migliorando l'affidabilità del sistema.

Diagramma delle sequenze

Il funzionamento di DataCollector e CircuitBreaker è anche visualizzabile dal seguente diagramma:



DataBase

Per quanto riguarda il DataBase abbiamo deciso di utilizzare MySql vista la maggior nostra familiarità con l'ambiente.

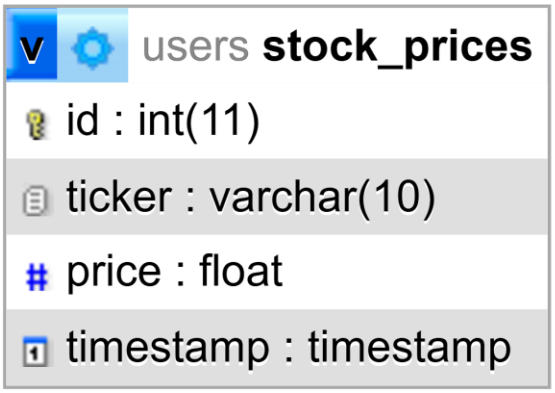
Per creare il container che permette di far girare il DB abbiamo usato l'apposita configurazione definita nel file *docker-compose.yml*.

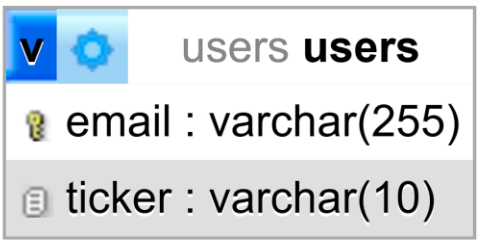
Dopo aver definito il suddetto file è bastato eseguire il comando:

```
docker-compose up –build
```

per creare e far partire tutti i servizi.

La struttura del DB è visibile anche nel seguente schema:





Container

Tutti i servizi descritti in precedenza (Server, DataCollector, CircuitBreaker e DataBase) sono eseguiti all'interno di container Docker. I container rappresentano unità leggere e autonome che contengono tutto il necessario per l'esecuzione di un'applicazione, inclusi codice, librerie, dipendenze e variabili di ambiente. Questa configurazione garantisce che l'applicazione funzioni in modo coerente in diversi ambienti, semplificando il processo di sviluppo e distribuzione.

Realizzazione del DockerFile

Per ogni servizio è stato necessario creare un Dockerfile. Il Dockerfile è un file di testo contenente una serie di istruzioni che permettono di assemblare l'immagine Docker del servizio. Ogni Dockerfile specifica:

- L'immagine di base da cui partire.
- I comandi per installare le dipendenze necessarie.

- La copia dei file sorgente dell'applicazione nell'immagine.
- L'impostazione delle variabili di ambiente necessarie.
- Il comando di esecuzione che avvia l'applicazione all'interno del container.

Creazione delle Immagini Docker

Utilizzando i Dockerfile, si possono assemblare le immagini Docker per ciascun servizio. Le immagini Docker rappresentano le basi da cui vengono eseguiti i container, assicurando che ogni container contenga tutto il necessario per l'esecuzione dell'applicazione specificata.

Configurazione della Rete tra Container

Per creare una rete tra i vari container, è necessario definire dei bridge. Questo processo è gestito attraverso un file *docker-compose.yml*. Questo file permette di:

- Definire i servizi e le relative immagini Docker.
- Configurare le reti e i bridge tra i container, consentendo la comunicazione tra i diversi servizi.
- Impostare le variabili di ambiente condivise.
- Specificare le dipendenze tra i servizi, garantendo un corretto ordine di avvio.

Un esempio di file docker-compose.yml potrebbe includere la definizione dei servizi, come database, server, datacollector e circuitbreaker, specificando le immagini da utilizzare o i Dockerfile da cui costruirle, le porte da esporre e le dipendenze tra i servizi.

Questa configurazione consente di avviare e gestire l'intera architettura dei container in modo semplice e coerente, semplificando il processo di sviluppo, test e produzione.

Client gRPC

Il client è stato realizzato per testare l'applicazione, il suo funzionamento prevede un interfaccia basica disponibile dal prompt dei comandi.

Si ha un menù dal quale si possono eseguire tutte le operazioni specificate nella consegna ed alcune operazione che abbiamo aggiunto per visualizzare velocemente lo stato del sistema.

Screen del menu del client

L'invocazione delle funzioni mediante gRPC e il file .proto descritto in precedenza è estremamente semplice; è simile a chiamare una funzione definita sulla stessa macchina del client. L'intera logica di connessione e trasmissione dei dati è resa trasparente per il client.