Progetto SDCC- Traccia B2

Luca Fiscariello 0318266

***Abstract*—Obiettivo del progetto è l'implementazione di tre algoritmi di mutua esclusione di cui due distribuiti e uno centralizzato. Il linguaggio di programmazione utilizzato è go. La descrizione del progetto nel seguente report è suddivisa grossomodo in due sezioni: nella prima sezione si discuteranno gli aspetti comuni a tutti gli algoritmi implementati, nella seconda invece si entrerà nel merito descrivendo dettagliatamente le caratteristiche di ogni algoritmo.**

I. Introduzione

Sono stati implementati i seguenti algoritmi per la gestione della mutua esclusione :

- Lamport distribuito;
- Token distribuito;
- Autorizzazione centralizzato;

Tutti e tre gli algoritmi implementano la gestione **dell'heartbeating.** In aggiunta gli algoritmi di token distribuito e autorizzazione centralizzata implementano ulteriori meccanismi per la gestione dei fallimenti.Tali meccanismi verranno discussi nel dettaglio in seguito. L'algoritmo distribuito di Lamport invece è stato realizzato nella versione base. Si è cercato di calare l'utilizzo di questi algoritmi in un contesto reale con l'obiettivo di creare un'applicazione funzionante. In particolar modo l'applicazione realizzata è **una pagina web che raccoglie le notizie aggiornate in tempo reale** relative a tematiche particolari. Ogni nodo della rete, a cui è assegnato una tematica specifica, vorrebbe pubblicare un insieme di notizie sulla pagina web, ma per fare ciò deve prima coordinarsi con gli altri nodi per evitare che ci siano scritture concorrenti.

II. Architettura generale

L'architettura dell'intera applicazione è l'insieme di 4 componenti: un **cluster di nodi** che operano in modo concorrente, il cluster **kafka** che fa da intermediario per la comunicazione tra i vari nodi, un unico **sender** che riceve articoli dai nodi concorrenti tramite grpc e li pubblica sulla pagina web tramite websocket e il **web server** che ospita la pagina web. Appare evidente come la sezione critica in questo contesto è rappresentata dalla chiamata alla procedura remota che avvia la scrittura di dati sulla pagina web. Tutti gli algoritmi implementati hanno come punto di partenza questo scenario. Quello che cambia tra un implementazione e un altra è il modo in cui i nodi della rete dialogano tra loro per trovare il modo di accedere alla sezione critica.

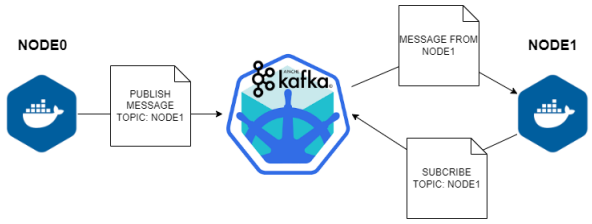

Fig. 1. Esempio di come avviene la comunicazione tra due nodi. Il NODE0 pubblica un messaggio sul topic NODE1. Il NODE1 si sottoscrive al topic che ha come nome il proprio id e rimane in ascolto di messaggi.

A. Cluster Nodi

I nodi sono le entità dell'applicazione che lavorano in modo concorrente e che devono essere opportunamente coordinati. Ogni nodo è identificato da un ID formato dalla stringa costante "node" a cui e concatenato un numero crescente. Non è obbligatorio che l'id dei nodi abbia esattamente questa struttura, la cosa fondamentale però è che nell'id di ogni nodo sia presente un numero che permetta di discriminare un nodo da un altro. Per comunicare i nodi sono tenuti a conoscere solo gli id degli altri nodi ma non la loro effettiva posizione (indirizzo ip e numero di porta) nel cluster. Questo perchè la comunicazione è mediata da kafka. Tale scelta comporta un grandissimo vantaggio: i nodi non dovranno gestire in prima persona le connessioni con le altre entità del cluster , ma sarà responsabilità di kafka inoltrare i messaggi ai corretti destinatari. Quando un nodo vorrà inviare un messaggio ad uno specifico destinatario non farà altro che inoltrare il messaggio a kafka chiedendo di pubblicarlo su uno specifico topic. Il destinatario a sua volta si sottoscriverà sullo stesso topic rimanendo in ascolto. Il topic in questione avrà come nome l'id del nodo destinatario. Tale meccanismo permette sia al mittente che al il destinatario di inviare e recuperare messaggi in maniera molto agevole. Questo concetto è chiarito nella **Fig. 1**.

B. Kafka

Il cluster di kafka rappresenta la componente centrale dell'applicazione attraverso cui tutti i nodi possono dialogare tra loro. Al primo avvio l'applicazione è configurata in modo da creare nel cluster di kafka dei topic "comuni" a cui tutti i nodi potranno leggere e pubblicare messaggi. I topic in questione sono utilizzati per la gestione dell'heart beating, comunicare l'avvio dell'applicazione, invio e ricezione di messaggi di presentazione. Si è preferito utilizzare kafka piuttosto che gestire la comunicazione tramite TCP poichè kafka è un

sistemsa p/s distribuito che offre quindi un disaccoppiamento spaziale e temporale. Inoltre permette di specificare un certo livello di replicazione dei dati offrendo una maggiore tolleranza ai guasti. Con queste scelte architetturali si rende inoltre possibile l'accesso alla rete di altri nodi in maniera dinamica. Un nodo che ha intenzione di unirsi alla rete dovrà pubblicare un messaggio di presentazione su un topic opportuno, per comunicare la sua presenza agli altri nodi della rete.Gli altri nodi rimarranno permanentemente in ascolto su questo topic di "presentazione" e sapranno immediatamente quando un nuovo nodo si unisce alla rete. L'unico difetto di questa soluzione è che ogni nodo è obbligato a conoscere tutti i membri della rete. Se la rete è piccola non ci sono particolari problemi, se è di dimensioni considerevoli ogni nodo dovrà gestire molte informazioni. Più nel dettaglio i topic kafka verranno utilizzati per :

- Heartbeating : Periodicamente un nodo *X* che è interessato a conoscere lo stato degli altri nodi della rete invia un messaggio di *heart* sul topic comune **heartbeat**. Attende successivamente l'arrivo dei messaggi di *beat* da tutti gli altri nodi. I vari messaggi di *beat* verranno ricevuti sul canale privato del mittente dell'*heart*. Il nodo *X* attenderà la ricezione dei *beat* fino allo scadere di un timeout al termine del quale aggiornerà le informazioni dei nodi attivi all'interno di una mappa che associa l'id del nodo al suo stato(attivo/non attivo). Questa mappa è particolarmente utile perchè il nodo potrà usarla per capire se contattare o meno un nodo in futuro.
- Presentation : Un nodo *X* che ha intenzione di entrare nella rete di nodi concorrenti invia un messaggio di presentazione sul topic comune **presentation** per avvisare tutti che è attivo nella rete. I nodi che ricevono questo messaggio memorizzano l'id del nodo *X* in una struttura dati. Lo stato di quel nodo verrà aggiornato in un secondo momento grazie all'heartbeating.
- Start applicazione: La comunicazione tra nodi può avere inizio solo in seguito all'apertura della pagina web da parte di un utente. Solo in quel momento è infatti possibile stabilire una connessione websocket tra il sender e il cliente javascript incapsulato nella pagina web. Non appena la connessione sarà stabilita verrà pubblicato un messaggio di *start* sul canale condiviso denominato appunto **start**. Tale messaggio serve per avvisare tutti i nodi che la connessione websocket è avvenuta e quindi che la cooperazione tra nodi concorrenti può avere inizio.
- Topic privati: Ogni nodo al momento dell'avvio chiederà anche la creazione di un canale "privato" in cui solo il nodo corrente potrà leggere ma tutti gli altri nodi della rete potranno scrivere. Tale topic avrà lo stesso nome dell'**ID del nodo**.

Un esempio che chiarisce l'utilizzo dei topic è presente alla **Fig. 2**.

C. Sender e API

Un'altra componente essenziale nell'applicazione è il Sender il quale riceve le richieste di scrittura sulla pagina web

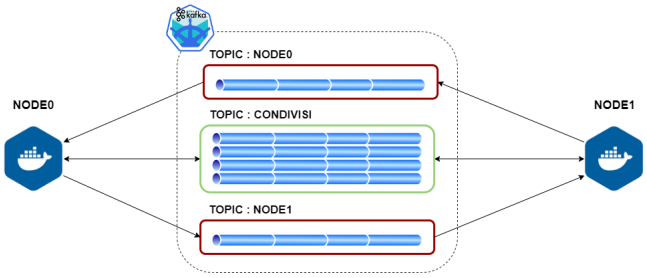

Fig. 2. La figura mostra come kafka gestisca sia i topic condivisi (in verde) che quelli privati (in rosso). In particolare ogni nodo potrà leggere e scrivere nei topic condivisi, leggere nel proprio topic e scrivere su tutti i topic degli altri nodi.

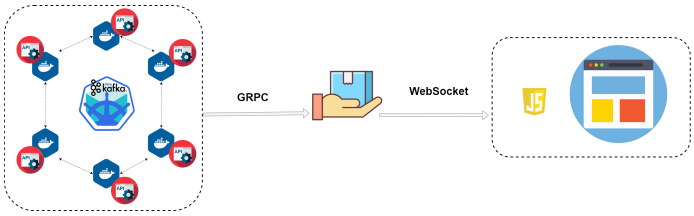

Fig. 3. La figura mostra l'architettura generale dell'applicazione. Un cluster di nodi si coordina attraverso kafka e scarica un certo numero di notizie da un API. Nel momento in cui un nodo accede in sezione critica invoca una procedura a chiamata remota richiedendo la pubblicazione di un articolo. Il sender soddisferà questa richiesta pubblicando sulla pagina web un articolo sfruttando la connessione websocket.

da parte degli altri nodi.Il Sender prevede inoltre un ulteriore metodo che permette di effettuare il refresh della connessione **websocket** nel momento in cui l'utente aggiorna la pagina oppure la chiude e successivamente la riapre. Il Sender opera attraverso un invocazione **grpc**, dalla quale riceve un messaggio contenente 4 informazioni: il titolo dell'articolo , una breve descrizione, un link ad un immagine di presentazione e il link all'articolo completo. Tutte queste informazioni, che verranno inserite in un'apposita card HTML, vengono fornite dai vari nodi che operano in modo concorrente e che scaricano queste informazioni da un API [1] . L'API offre la possibilità di specificare la lingua degli articoli ma anche un tema da ricercare. Il grande vantaggio di poter usare una connessione web socket è che la pagina può essere aggiornata dinamicamente senza dover effettuare nessun ricaricamento. Purtroppo il canale di comunicazione websocket non supporta la concorrenza ed è proprio per questo che è necessario coordinare tutti i nodi.Un esempio più chiaro dell'architettura è presentata nella **Fig. 3**.

III. Deployment

Il deployment dell'applicazione avviene tramite docker compose. L'idea è quella di realizzare una rete di container che dialogano tra loro. Tra i vari container istanziati 2 di essi contengono *zookkeeper* e un *broker kafka*. Ogni nodo verrà poi ospitato in un container differente. Per configurare l'applicazione è sufficiente modificare il *docker-compose.yaml* e indicare quanti nodi si vogliono creare e quale tema si vuole associare a ciascuno di essi. La rete di container

è stata a sua volta deployata su un istanza di *ec2* sfruttando *ansible*. In particolare è stato creato un file di configurazione che una volta eseguito permette di scaricare il codice dell'applicazione da un repository github , salvarlo su un'istanza di *ec2* specificata e lanciare l'applicazione. Per far partire in automatico l'applicazione all'avvio dell'istanza *ec2* è stato creato un servizio , memorizzato opportunamente con *ansible* sull'istanza *ec2*, che ha come obiettivo quello di istanziare la rete di container.

IV. Riferimenti codice

Tutti gli algoritmi(applicazioni) sono implementati avendo una base di codice comune.Tutto il codice prodotto è salvato su un repository github. La cartella contenente il codice di un solo algoritmo si compone di due directory principali:

- *../Node* : Contiene la logica di gestione di un nodo della rete. Il codice che avvia il nodo è *../Node/main*. Nella cartella *../Node* sono presenti altri moduli, il più importante dei quali è *../Node/handler*. Gli handler implementati e comuni a tutti gli algoritmi sono:
  - APIHandler : ha l'obiettivo di scaricare gli articoli di un determinato tema tramite l'API. Questi articoli verranno poi inviati dal programma principale (*../Node/main*) al Sender.
  - HeartBeatHandler : ha il compito di gestire l'heartbeating.
  - KafkaHandler : ha il compito di gestire alcune interazioni tra il nodo corrente e il broker del cluster kafka. In particolare vengono modellate due funzioni che permettono la creazione di un nuovo topic e l'invio di un messaggio di presentazione sul topic*presentation*.
  - NodeActiveHandler : ha il compito di memorizzare quali sono i nodi della rete attualmente attivi. Queste informazioni sono memorizzate in una mappa. Tale handler verrà sfruttato prevalentemente dall'handler dell'heartbeat il quale dopo aver completato un ciclo di heart-beat aggiornerà i dati nella mappa.
- *../Webserver* : Ospita l'implementazione del web server che hosta la pagina web. Al path *../Webserver/grpc/implementation* è presente l'implementazione del sender.

V. Token distribuito

A. Descrizione generale

Tale algoritmo permette di discriminare l'accesso alla sezione critica tramite il possesso di un token. Logicamente i nodi sono organizzati in una rete ad anello. Il nodo *X* che possiede il token uscito dalla sezione critica lo girerà al nodo *X+1*. Se il nodo *X+1* non è attivo allora il token varrà inoltrato al nodo ancora successivo(*X+2*). Questo procedimento può continuare in modo iterativo e ciclico perchè ogni nodo conosce tutti gli altri nodi della rete. Per capire quale è l'identificativo del nodo a cui inoltrare il token il nodo analizza il proprio id e somma poi 1 ad esso. In ogni istante il nodo che vuole inviare il token conosce quali altri nodi della rete sono attivi grazie al meccanismo dell'hearbeating. I nodi non attivi non verranno quindi contattati. Potrebbe comunque verificarsi che il nodo che dispone del token vada in crush. In quel caso è stato implementato un ulteriore meccanismo per risolvere i guasti permanenti ma non quelli temporanei che potrebbero causare una duplicazione del token.

B. Gestione fallimenti

Come è generato il token e come vengono contrastati i fallimenti non temporanei? La soluzione trovata consiste nell'eleggere un nodo leader all'interno della rete. Il nodo con id più piccolo verrà eletto leader. Si sceglie il nodo con id più piccolo poichè si vuole che il leader venga eletto una sola volta e rimanga attivo finchè non subisce un fallimento(o finchè non terminerà l'applicazione). Se fosse stato implementato l'algoritmo bully nella versione tradizionale , ad ogni nuovo accesso di un nodo nella rete, si verificherebbe una nuova elezione. Ad ogni modo, il leader in seguito all'elezione, assume la funzione di token creator e token checker. Il leader sarà infatti il primo nodo a creare il token e distribuirlo. Ma il suo lavoro non termina qui. Dopo aver rilasciato il token resta in attesa di messaggi di ack degli altri nodi. Ogni volta che un nodo riceverà il token ,nel normale ciclo di funzionamento dell'applicazione, contatterà il leader per informarlo che nella rete il token circola correttamente. Quando il leader non riceverà più token per un tempo molto lungo darà per scontato che il token è andato perso e quindi ne genererà uno nuovo. Proprio per questo motivo se un nodo possessore del token va subisce un fallimento momentaneo , e questo fallimento si protrae oltre il timeout del checker, potrebbe verificarsi una replicazione del token. Un esempio che chiarisce come i nodi si coordinano tra loro è presentato nella **Fig. 4**.

C. Vantaggi e svantaggi

Il primo vantaggio dell'algoritmo è la fairness. Essendo i nodi organizzati in una strutturato ad anello prima o poi ogni nodo riceverà il token. Questa versione risulta inoltre robusta ai guasti permanenti , ma anche a guasti temporanei che non superano il timeout del token checker. Inoltre il fatto di implementare heartbeating offre ulteriori garanzie: un nodo che risulta down non riceverà mai il token non potendo causare in questo modo un blocco della rete . Lo svantaggio di questa implementazione è che si consuma comunque banda facendo circolare ciclicamente il token anche quando qualche nodo non è interessato a pubblicare articoli. Va anche considerato il fatto che guasti momentanei che sono maggiori del timeout del checker comportano una duplicazione del token.

D. Riferimenti codice

Rispetto agli altri algoritmi in questo caso sono stati implementati altri handler.

- TokenCheckerHandler : ha il compito di supervisionare la comunicazione e assicurarsi che nodi continuino a comunicare senza entrare in deadlock. Ha inoltre il compito

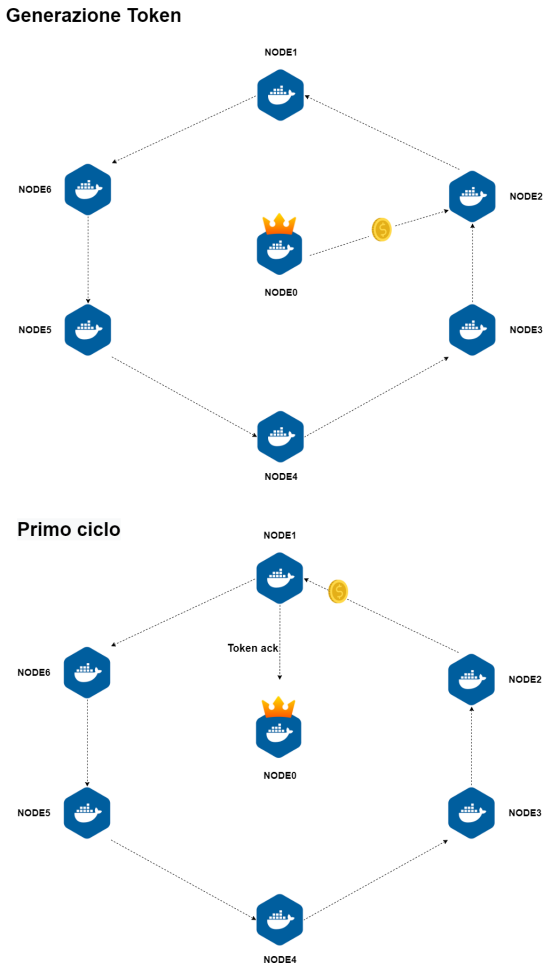

Fig. 4. Nell'immagine viene riportata la fase iniziale dell'algoritmo. Il nodo con id più piccolo è eletto leader. Il leader genererà il token e lo inoltrerà nella rete. Il nodo che riceve il token lo invia al nodo successivo il quale comunicherà al leader che ha ricevuto correttamente il token.

di verificare che un token sia in circolo nella rete. Se il token è perso ne genera uno nuovo.

- TokenHandler : ha il compito di gestire la ricezione di l'invio del token dal nodo corrente a uno nuovo .

VI. Lamport distribuito

A. Descrizione generale

Tale algoritmi per gestire l'accesso alla sezione critica utilizza un clock scalare e una lista di richieste di accesso alla sezione critica mantenuta localmente da ogni nodo. Quando un nodo vuole entrare in sezione critica invia una richiesta a tutti i nodi della rete associando il timestamp e il proprio id. I nodi che ricevono la richiesta la memorizzano nella propria lista locale e inviano un ack. Un nodo entrerà in sezione critica se il suo timestamp è il più piccolo di tutti. Se due timestamp sono uguali entrerà il nodo con id più piccolo. Quando un nodo esce dalla sezione critica invece, invia un messaggio di rilascio a tutta la rete in modo che tutti i nodi possano cancellare la richiesta appena soddisfatta dalla propria lista locale.

B. Gestione fallimenti

L'algoritmo è implementata nella versione base. Non presenta ulteriori meccanismi per la gestione dei fallimenti. Questo vuol dire che se un nodo dovesse fallire allora tutta l'applicazione andrebbe in deadlock. L'heart beating è comunque implementato ed offre una debole garanzia. Un nodo che richiede l'accesso alla sezione critica attende gli ack solo dei nodi attivi al momento dell'invio della richiesta. Potrebbe però verificarsi uno scenario non favorevole. Si ipotizzi una situazione in cui un nodo *X* invia una richiesta e tutti i nodi della rete in quel preciso istante sono attivi. Un ipotetico nodo *X+1* potrebbe andare però in fallimento prima di riuscire a inviare l'ack. Il nodo *X* tuttavia si aspetta di ricevere tanti ack quanti sono i nodi della rete. Ovviamente questo non accadrà e l'applicazione entrerà in deadlock. L'Heartbeating quindi offre una debole garanzia ma non sufficiente.

C. Vantaggi e svantaggi

Il vantaggio della soluzione è l'estrema semplicità. Richiede lo scambio di solo 3 tipologie di messaggi. Inoltre risulta essere fair,safe e garantisce la liveness. Riesce anche a sopportare debolmente guasti momentanei e permanenti.

D. Riferimenti codice

Rispetto agli altri algoritmi in questo caso sono stati implementati altri handler.

- RequestHandler : Si occupa di aggiornare il clock logico e di mantenere la lista degli accessi in sezione critica. Invia richieste di accesso alla sezione critica, attende in modo bloccante tutti gli ack, invia ack e rilascia l'accesso alla sezione critica.

VII. Autorizzazione centralizzato

A. Descrizione generale

Tale algoritmo prevede l'esistenza di un nodo leader che concede autorizzazioni a chi richiede di entrare in sezione critica. Anche in questo caso sarà il nodo con id più basso ad essere eletto leader. Un nodo che vuole accedere alla sezione critica invia una richiesta al leader, attende in modo bloccante la sua autorizzazione e infine una volta uscito dalla sezione critica invia un messaggio di rilascio. Il leader organizza le richieste in maniera FIFO all'interno di una lista sulla base dell'ordine di arrivo.

B. Gestione fallimenti

Quando un nodo ottiene l'accesso alla sezione critica deve inviare un messaggio di ack al leader. Se questo messaggio di ack non è ricevuto entro un certo timeout allora il nodo verrà considerato come non attivo. Un ulteriore meccanismo implementato è che il leader attende il messaggio di rilascio entro un timeout. Se il messaggio di rilascio non è ricevuto allora l'autorizzazione verrà sottratta e un nuovo nodo potrà entrare in sezione critica. Di base ovviamente il leader prima di concedere l'autorizzazione controllerà se il nodo è ancora attivo osservando la mappa aggiornata costantemente dall'heartbeating. Sono stati presi infine provvedimento anche

per l'elezione del leader : un nodo *X* ricalcola periodicamente l'id del leader. Se l'id del leader è quello del nodo corrente vuol dire che leader è andato in down e il nodo corrente deve prendere il suo posto.

C. Vantaggi e svantaggi

Il vantaggio di questa soluzione è la semplicità in quanto ogni nodo deve inviare solo 3 messaggi. Poichè i messaggi scambiati sono pochi si consumerà poca banda.L'algoritmo è safe e fair. Inoltre viene meno anche il single point of failure in quanto ci saranno nuove elezioni di leader in caso di fallimenti. Sono supportati guasti anche sufficientemente lunghi del leader. Quando il leader va in down e ritorna, tutti gli altri nodi lo notano e contatteranno lui per le successive coordinazioni. Un problema di questa implementazione è che se un nodo è attivo e tarda a rilasciare l'autorizzazione , un altro nodo potrà accedere alla sezione critica creando accessi concorrenti.

D. Riferimenti codice

Rispetto agli altri algoritmi in questo caso sono stati implementati altri handler.

- LeaderHandler : Questo handler verrà utilizzato dal nodo della rete eletto leader che dovrà gestire l'accesso alla sezione critica tramite autorizzazioni. L'handler implementa due funzionalità per:mettersi in ascolti di messaggi di richiesta di degli altri nodi e mettersi in ascolti di messaggi di release. Il leader prima di rilasciare l'autorizzazione si assicura che il nodo richiedente sia ancora attivo attendendo un suo ack. Allo scadere di un timeout se il nodo che aveva l'accesso alla sezione critica non rilascia l'accesso gli viene sottratto in quanto considerato non attivo.
- HandlerLeaderComunication: gestisce la comunicazione con tra nodo corrente e il leader. In particolare l'Handler implementa funzionalità per: richiedere accesso alla sezione critica al leader, inviare un ack in seguito alla ricezione dell'autorizzazione del leader, inviare messaggio di rilascio quando il nodo corrente ha intenzione di uscire dalla sezione critica.

VIII. Risultato finale

Indipendentemente dall'algoritmo utilizzato, una volta avviata l'applicazione(in locale o eventualmente in remoto tramite l'istanza di *ec2*) è possibile osservare come dinamicamente il sito venga popolato da notizie. La **Fig. 5** mostra un possibile output dell'applicazione.

Fig. 5. Esempio dell'output dell'applicazione.

References

[1] API rest, http://newsapi.org. Gratuita per uso privato.