Implementazione distribuita dell'algoritmo Flood-Fill

Tam Gabriele, Merlo Filippo

13 novembre 2024

Indice

Sommario

L'algoritmo Flood-Fill è una tecnica utilizzata per ricolorare aree connesse in strutture bidimensionali, comunemente applicato in grafica computerizzata. L'obiettivo di questo progetto è l'implementazione distribuita dell'algoritmo, con una rappresentazione dove ogni pixel di un'immagine è trattato come un nodo di una rete distribuita. In questo contesto, emergono sfide legate alla comunicazione tra nodi, alla consistenza dello stato, alla gestione delle richieste concorrenti e alla tolleranza ai guasti.
Nel documento viene proposto un approccio basato su una partizione dei nodi in cluster, gestiti da nodi leader, e su una rete overlay tra leader per facilitare operazioni di merge e coordinamento. Viene implementato un sistema che garantisce scalabilità, robustezza e consistenza utilizzando un time-server che gestisce i timestamp per l'ordinamento degli eventi e un database condiviso per la sincronizzazione globale. Inoltre, il sistema è progettato per essere resiliente ai guasti, con meccanismi di elezione di nuovi leader e recupero dei nodi. I risultati mostrano che la soluzione proposta è efficiente e adatta a reti di grandi dimensioni.

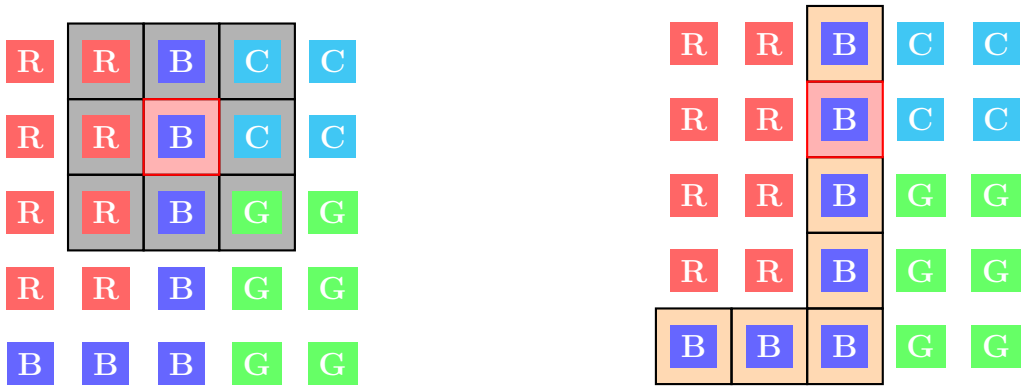
Capitolo 1

Introduzione

In questo capitolo viene descritto il problema principale affrontato dal progetto, insieme a una panoramica della soluzione proposta. L'obiettivo è sviluppare una versione distribuita dell'algoritmo *Flood-Fill*, che permette di ricolorare aree connesse in una struttura bidimensionale, come un'immagine digitale, dove ogni pixel è rappresentato da un nodo della rete distribuita. Questo scenario introduce sfide legate alla comunicazione tra nodi, alla gestione delle operazioni concorrenti e alla tolleranza ai guasti, tutti aspetti che devono essere gestiti in modo efficiente.

1.1 Descrizione del problema

Il problema consiste nel ricolorare un'area connessa di una matrice bidimensionale $img[N][M]$, dove ogni elemento $img[i][j]$[1] rappresenta il colore di un pixel in posizione (i, j). Dato un pixel iniziale in posizione `(x, y)` e un nuovo colore `newColor`, l'obiettivo è cambiare il colore del pixel selezionato e di tutti i pixel adiacenti che condividono lo stesso colore iniziale con `newColor` (vedi Fig. 1.1). Questa ricolorazione deve propagarsi attraverso nodi adiacenti, in tutte le direzioni (orizzontale, verticale, diagonale).



(a) L'area evidenziata in grigio sono i nodi adiacenti.

(b) L'area evidenziata in giallo sono i nodi del cluster.

Figura 1.1: A partire dal nodo scelto (evidenziato di rosso), si mostrano i suoi nodi adiacenti (1.1a) e quelli del cluster (1.1b).

Ogni nodo della rete è responsabile di un pixel e collabora con i nodi adiacenti per completare l'operazione di ricolorazione.

[1] con $0 \leq i < N$ e $0 \leq j < M$

1.2 Terminologia

Per una migliore comprensione e per evitare ambiguità, vengono definiti alcuni termini chiave:

Nodo: un processo autonomo nella rete, responsabile di un singolo pixel dell'immagine.

Colore: la proprietà associata a ciascun nodo, che rappresenta lo stato corrente (colore) del pixel.

Nodi adiacenti: i nodi che si trovano in posizioni fisicamente adiacenti nella matrice, ovvero collegati orizzontalmente, verticalmente o diagonalmente.

Cluster: un gruppo di nodi, i quali condividono lo stesso colore e sono connessi tra loro. Nel nostro approccio di soluzione, i cluster vengono gestiti a livello locale per ottimizzare le operazioni di ricolorazione.

Nodo Leader: il nodo designato all'interno di un cluster per coordinare le operazioni globali, come l'avvio delle ricolorazioni e la fusione con altri cluster.

Ricolorazione/Cambio colore: l'operazione di aggiornamento del colore di un cluster.

Unione/Merge di cluster: l'unione di due o più cluster in uno singolo, nel caso in cui questi siano adiacenti e condividano lo stesso colore. Quest'operazione avviene in seguito a un cambio colore.

1.3 Struttura complessiva dell'implementazione

Il sistema è strutturato come una rete di nodi, i quali sono organizzati in cluster basati sul colore condiviso. Ogni cluster è gestito da un *nodo leader*, il quale si occupa di coordinare le operazioni sia interne che esterne al cluster: la gestione delle operazioni distribuite richiede una comunicazione efficiente, che è garantita tramite una rete overlay tra i leader dei cluster e l'uso di un server centrale per mantenere lo stato globale del sistema.

1.4 Caratteristiche del sistema distribuito

Il sistema implementa diverse funzionalità proprie dei sistemi distribuiti:

- **Trasparenza nella comunicazione**: ogni nodo comunica direttamente con i nodi adiacenti, o con il leader del cluster, senza necessità di conoscere la topologia completa.
- **Consistenza**: la consistenza globale del sistema è garantita tramite l'associazione di ogni operazione a un record evento. Gli eventi garantiscono sia una corretta gestione delle operazioni, tramite l'ordinamento per **timestamp** ed **id**, sia la loro archiviazione tramite logging. Vengono evitati così possibili conflitti dovuti alla concorrenza.
- **Tolleranza ai guasti**: meccanismi di failover assicurano che il sistema possa continuare a funzionare anche in caso di guasti di singoli nodi (sia normali che leader). I processi di ogni nodo vengono monitorati in caso di arresti inaspettati e riavviati immediatamente.

1.5 Algoritmi implementati

Il sistema utilizza vari algoritmi per garantire la corretta esecuzione dell'algoritmo distribuito di Flood-Fill:

- **Algoritmo di cambio colore**: i nodi comunicano al leader del cluster la volontà di cambiare colore. Quest'ultimo gestisce le richieste per mezzo di una coda e comunica ognuna al server per avere la conferma per procedere con la ricolorazione locale. Il server viene coinvolto sia per garantire la consistenza, sia per evitare comportamenti inaspettati del sistema nel caso di operazioni concorrenti in cluster differenti.
- **Algoritmo di merge dei cluster**: quando due cluster adiacenti condividono lo stesso colore (in seguito a operazioni di cambio colore), i leader coinvolti ed il server collaborano per eseguire l'operazione di merge.
- **Algoritmo di gestione dei fallimenti**: include procedure per la rielezione dei leader e la gestione dei nodi disconnessi o falliti.

1.6 Piano di test

Il sistema sarà testato simulando scenari di ricolorazione su immagini di grandi dimensioni: i test valuteranno la robustezza del sistema in differenti situazioni di concorrenza (concorrenza di cambio colore intra/extra-cluster e del merge di cluster) ed anche in presenza di guasti di nodi. L'attenzione sarà posta sulla consistenza delle operazioni distribuite.

1.7 Programma di sviluppo

Il progetto si articola in diverse fasi:

1. **Fase di progettazione**: definizione dell'architettura del sistema e degli algoritmi distribuiti.
2. **Fase d'implementazione**: sviluppo del codice del sistema distribuito nel linguaggio Erlang e scrittura del frontend e degli script di test in Python.
3. **Fase di testing**: esecuzione di test per verificare il corretto funzionamento del sistema, nonchè valutazione delle prestazioni e della scalabilità del sistema sviluppato.

Capitolo 2

Analisi

In questo capitolo vengono descritti nel dettaglio i requisiti funzionali e non funzionali di una soluzione per l'implementazione distribuita dell'algoritmo *Flood-Fill*. Questi requisiti guidano lo sviluppo del sistema, specificando le funzionalità richieste e le qualità desiderabili, insieme alle assunzioni e ai vincoli considerati durante l'implementazione.

2.1 Requisiti Funzionali

I requisiti funzionali definiscono le operazioni principali che il sistema deve offrire, specificando come devono essere gestiti input e output, e l'effetto atteso per ogni funzione.

1. **Operazione di colorazione distribuita**: il sistema deve permettere a un nodo di iniziare un'operazione di ricolorazione, propagando il nuovo colore a tutti i nodi adiacenti che condividono lo stesso colore iniziale.

2. **Gestione di richieste concorrenti**: il sistema deve essere in grado di gestire più richieste di ricolorazione simultanee provenienti da nodi diversi, garantendo un comportamento deterministico e la corretta risoluzione dei conflitti.

3. **Comunicazione tra nodi**: i nodi devono essere in grado di scambiarsi messaggi con i loro vicini adiacenti per la sincronizzazione dello stato e la coordinazione delle operazioni. Ogni nodo deve mantenere aggiornate le informazioni relative ai vicini e al leader del proprio cluster.

4. **Unione di cluster**: quando due o più cluster adiacenti diventano dello stesso colore in seguito a ricolorazioni, il sistema deve unificarli in uno solo.

5. **Tolleranza ai guasti**: il sistema deve essere in grado di gestire il fallimento di singoli nodi, sia leader che normali, garantendo che il resto del cluster continui a operare correttamente. Devono essere previsti meccanismi di failover per la rielezione dei leader.

6. **Consistenza globale**: il sistema deve mantenere una consistenza globale dello stato, garantendo che tutti i nodi di un cluster condividano lo stesso colore dopo ogni operazione di cambio colore o merge.

2.2 Requisiti Non Funzionali

I requisiti non funzionali riguardano le qualità desiderabili del sistema, tra cui prestazioni, sicurezza e tolleranza ai guasti. Essi descrivono le caratteristiche che devono essere garantite durante l'implementazione.

1. **Scalabilità**: il sistema deve essere in grado di supportare un elevato numero di nodi, mantenendo buone prestazioni anche con una rete distribuita estesa.
2. **Efficienza delle comunicazioni**: il numero di messaggi scambiati tra i nodi deve essere minimizzato per evitare sovraccarichi di rete. Gli algoritmi di ricolorazione e unione devono utilizzare una comunicazione locale, limitata ai vicini diretti, salvo necessità di sincronizzazione globale.
3. **Robustezza e affidabilità**: il sistema deve essere resiliente ai guasti e garantire il recupero rapido dopo il fallimento di uno o più nodi, inclusi i nodi leader. Deve essere possibile continuare le operazioni anche in caso di guasti isolati.
4. **Consistenza**: il sistema deve garantire la consistenza dello stato anche in presenza di richieste concorrenti o di fallimenti.

2.3 Assunzioni e Vincoli

Le seguenti assunzioni e vincoli sono stati considerati per l'implementazione del sistema distribuito:

- **Topologia della rete**: si assume che la rete sottostante sia affidabile, ossia che i messaggi inviati tra nodi adiacenti vengano consegnati correttamente entro un tempo limite ragionevole.
- **Fallimenti dei nodi**: si assume che i nodi possano fallire in modalità *crash*, ossia che possano smettere di funzionare senza comportamento bizantino (comportamenti non affidabili o malevoli).
- **Conoscenza locale**: ogni nodo ha una conoscenza locale limitata dei propri vicini e del leader del proprio cluster. Nessun nodo ha una visione globale dell'intera rete o di tutti i cluster.
- **Inizializzazione dei nodi**: al momento dell'avvio, ogni nodo conosce solo le proprie coordinate all'interno della matrice dell'immagine e il colore iniziale del pixel che rappresenta. La formazione dei cluster e la gestione della rete overlay avvengono durante la fase iniziale d'avvio del sistema.

Capitolo 3

Progettazione del sistema

In questo capitolo viene presentata la progettazione del sistema: verranno analizzate le possibili soluzioni, spiegando i vantaggi e gli svantaggi di ciascuna e successivamente verrà presentata la soluzione scelta, descrivendone architettura, componenti ed algoritmi utilizzati.

3.1 Analisi delle possibili soluzioni

Sono stati considerati due principali approcci:

1. **Comunicazione diretta tra nodi**: nodi normali e leader comunicano direttamente tra loro per eseguire le operazioni di cambio colore e merge. Cluster adiacenti comunicano direttamente per mezzo dei propri leader.
2. **Partizionamento del grafo e elezione di nodi leader**: il grafo viene suddiviso in cluster, ciascuno gestito dal proprio nodo leader che ne coordina operazioni (e.g. ricolorazione) ed organizzazione (e.g. fallimento di nodi, merge, ...).

1. Comunicazione diretta tra nodi

Descrizione generale

In questo approccio, ogni nodo mantiene informazioni locali e comunica direttamente con i nodi adiacenti. Quando un nodo cambia colore, l'operazione di ricolorazione viene unicamente propagata ai nodi adiacenti che condividono lo stesso colore, creando una diffusione a cascata.

Vantaggi

- **Semplicità**: l'implementazione è relativamente semplice poiché ogni nodo gestisce solo informazioni locali.
- **Assenza di punti singoli di fallimento**: non esistono nodi centrali il cui fallimento compromette l'intero sistema.

Svantaggi

- **Scalabilità limitata**: l'elevato numero di messaggi scambiati può diventare insostenibile in reti di grandi dimensioni.
- **Concorrenza non deterministica**: difficoltà nel gestire richieste concorrenti in modo deterministico.
- **Gestione dei fallimenti complessa**: la mancanza di una visione globale rende difficile il recupero da guasti.

2. Partizionamento del grafo e nodi leader

Descrizione generale

In questo approccio, il grafo è partizionato in cluster, ognuno dei quali è gestito dal proprio **nodo leader** responsabile della gestione dello stato e delle decisioni per il cluster. I nodi normali delegano le decisioni al leader, riducendo la complessità locale.

Vantaggi

- **Consistenza migliorata**: la centralizzazione delle decisioni nel leader migliora la consistenza del sistema.
- **Gestione efficiente della concorrenza**: il leader gestisce richieste concorrenti in modo ordinato.
- **Scalabilità**: partizionando il grafo si riduce la complessità in cluster più piccoli.

Svantaggi

- **Punti singoli di fallimento**: il fallimento del leader può compromettere il funzionamento del cluster.
- **Complessità aggiuntiva**: l'implementazione di leader, algoritmi di elezione e meccanismi di consenso aggiungono complessità.
- **Possibili colli di bottiglia**: il leader può diventare un collo di bottiglia in cluster molto attivi.

3.2 Soluzione proposta

In questo progetto è stato usato come base il secondo approccio appena presentato, il quale è stato arricchito di varie scelte implementative atte a limitare al minimo gli svantaggi della soluzione, senza intaccare i vantaggi di essa. L'idea della soluzione qui presentata è di trattare ogni cluster come un nodo in un grafo *overlay*: ciò permette una gestione efficiente delle comunicazioni tra cluster, semplifica le operazioni di merge e cambio colore, e facilita la consistenza globale del sistema.

3.2.1 Idea principale

L'idea centrale della soluzione è la seguente:

- **Trattamento dei cluster come nodi in un grafo overlay**: ogni cluster viene considerato come un singolo nodo in un grafo overlay, dove i nodi rappresentano i leader dei cluster e gli archi rappresentano le adiacenze tra cluster.
- **Comunicazione tra leader**: i nodi leader mantengono una lista dei leader adiacenti nel grafo overlay, permettendo comunicazioni dirette per le operazioni di merge e coordinamento.

- **Utilizzo di un server centrale per la sincronizzazione**: implementazione di un server condiviso per gestione, ordinamento e memorizzazione delle operazioni eseguite. Facilita la consistenza e la gestione delle operazioni concorrenti, fornendo al tempo stesso ridondanza sia delle operazioni eseguite (attraverso un file di log) sia dello stato globale della rete.

3.2.2 Setup iniziale

La fase di setup è fondamentale per l'inizializzazione dei parametri dei nodi e la costruzione dei cluster, i quali costituiranno il sistema distribuito. Poiché i nodi inizialmente conoscono solo informazioni su loro stessi, il processo di formazione dei cluster e la scoperta delle connessioni tra di essi viene suddiviso in tre fasi, ciascuna con uno scopo specifico:

- **Fase 1: Scoperta dei vicini**
- **Fase 2: Formazione dei cluster**
- **Fase 3: Scoperta dei cluster adiacenti e costruzione della rete overlay**

Inizialmente ogni nodo conosce solo:

- Le proprie coordinate `(x, y)` nell'immagine.
- Il proprio colore `color`, assegnato randomicamente da una palette predefinita oppure impostato dall'utente attraverso un file contente la lista dei colori.

Fase 1: Scoperta dei vicini Attraverso le coordinate del nodo, vengono calcolate le posizioni dei nodi adiacenti. Con queste vengono recuperati gli id dei nodi, i quali vengono salvati nella lista *neighbors*.

Fase 2: Formazione dei cluster Per ogni nodo del sistema viene eseguito un algoritmo di scoperta dei nodi del cluster simile all'algoritmo BFS[1]. Ad ogni nodo viene associata una variable *Visited* inizialmente impostata a `False`. Sfruttando la lista *neighbors* creata nella fase precedente, vengono ricorsivamente scoperti i nodi adiacenti tra loro che condividono lo stesso colore. L'algoritmo di visita termina quando non ci sono più vicini con colore uguale a quello del nodo di partenza. Ogni nodo che inizia l'algoritmo di visita diventa leader del cluster, mentre tutti i nodi scoperti saranno nodi normali che fanno parte del cluster. Ogni nodo del cluster viene salvato nella lista *cluster_nodes*.

Fase 3: Scoperta dei cluster adiacenti e creazione della rete overlay Nella terza ed ultima fase, viene costruita la lista dei cluster adiacenti, in cui ogni elemento della lista è una tripla del tipo `{pid, colore, leader}`. La lista (*adj_nodes*) viene costruita facendo la sottrazione della lista dei nodi del cluster (*cluster_nodes*) dalla lista cumulativa di tutti i vicini (*neighbors*) di ogni nodo. La lista ottenuta è costituita da tutti i nodi che compongono il "contorno" del cluster, ovvero nodi vicini ma con colore diverso. Infine per ogni nodo nella lista vengono prese le informazioni riguardo all'id del leader e il colore del cluster, rimuovendo possibili duplicati e salvando tutto nella lista *adj_clusters*.

[1] Breadth-First Search

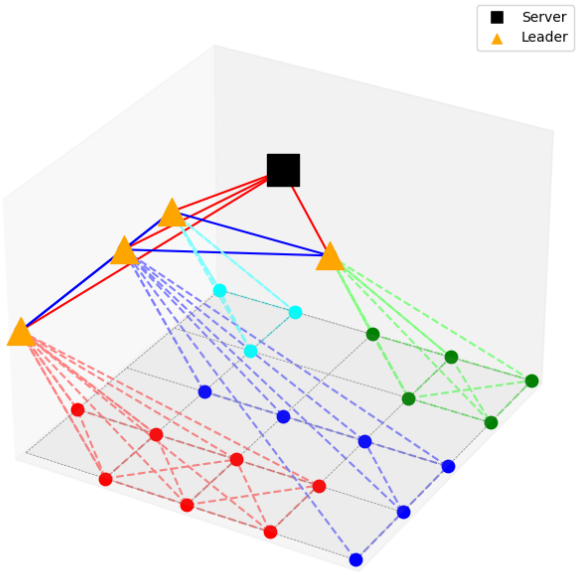


Figura 3.1: Esempio di grafo G con struttura a cluster e overlay tra i leader. Nel livello più basso sono presenti i nodi normali, disposti su una griglia 5x5 e colorati in base al cluster di appartenenza. Ogni cluster ha un leader, rappresentato da un triangolo arancione, elevato ad un livello superiore. I leader comunicano tra loro attraverso un overlay, mostrato con archi blu, e sono collegati al nodo server, rappresentato da un quadrato nero, mediante archi rossi.

Conclusione del setup Al termine delle 3 fasi di setup, ogni nodo non leader viene trasformato in nodo normale rimuovendo le informazioni non più necessarie, ma mantendendo solo

- `pid`: l'id del processo nodo.
- `(x, y)`: le proprie coordinate.
- `leaderID`: l'identificatore del leader del proprio cluster.
- `neighbors`: la lista dei nodi fisicamente vicini (indipendentemente dal colore).

Questo processo riduce l'overhead e ottimizza la memoria utilizzata dai nodi.
Al contrario, ogni leader comunica le informazioni del proprio cluster al server e quest'ultimo si occuperà di salvare lo stato globale del sistema.
Il sistema è ora sincronizzato ed è pronto per entrare nella fase operativa.

Complessità della fase di setup Di seguito viene riportata l'analisi della complessità delle due fasi.

- **Fase 1: Scoperta dei vicini**
  - Per ogni nodo vengono calcolati i vicini usando la propria posizione, pertanto vengono eseguite $N \times M$ chiamate.
  - Ciascuna chiamata ha complessità $O(8)$, in quanto vi possono essere al massimo 8 vicini considerando orizzontale, verticale e diagonale.

  - La complessità totale della fase 1 è quindi $O(N \times M)$.

- **Fase 2: Formazione dei cluster**
  - Il server avvia l'algoritmo per formare i cluster, scorrendo l'immagine da sinistra a destra e dall'alto verso il basso.
  - Per ogni nodo non ancora visitato, viene avviato un algoritmo di scoperta che include tutti i nodi adiacenti con lo stesso colore.
  - La complessità dell'algoritmo per ogni cluster è lineare rispetto al numero di nodi nel cluster, ossia $O(k)$, dove k è il numero di nodi del cluster.
  - Poiché ogni nodo dell'immagine viene visitato esattamente una volta, la complessità totale della formazione dei cluster è $O(N \times M)$, dove $N \times M$ rappresenta il numero totale di nodi nell'immagine.

- **Fase 3: Scoperta dei cluster adiacenti**
  - Dopo la formazione dei cluster, i leader inviano richieste ai propri nodi per verificare se i loro vicini appartengono a un altro cluster.
  - Ogni nodo comunica solo con i suoi vicini immediati, il che implica una complessità costante per ogni nodo, pari a $O(1)$.
  - La complessità totale della scoperta dei cluster adiacenti è anch'essa $O(N \times M)$, poiché ogni nodo partecipa al processo una sola volta.

- **Rete overlay e sincronizzazione finale**
  - Una volta scoperti i cluster adiacenti, viene costruita una rete overlay tra i leader.
  - Ogni leader comunica con il server e con i leader dei cluster adiacenti.
  - La complessità di questa fase è proporzionale al numero di cluster, ossia $O(C)$, dove C è il numero totale di cluster.

- **Conclusione sulla complessità**
  - La complessità totale della fase di setup è dominata dal numero di nodi nell'immagine, quindi è $O(N \times M)$.
  - Sebbene la costruzione della rete overlay abbia una complessità aggiuntiva $O(C)$, essa è limitata rispetto al termine dominante $O(N \times M)$.
  - Pertanto, la fase di setup risulta essere efficiente e scalabile, anche per immagini di grandi dimensioni.

3.2.3 Architettura del sistema

Nodi

Il sistema è composto da due tipi principali di nodi:

1. **Nodi normali**: rappresentano i pixel dell'immagine e mantengono informazioni minime. Ogni nodo normale conosce:
   - L'id del proprio processo `pid`.

- Le proprie coordinate `x, y` all'interno dell'immagine.
- Il proprio `leaderID`, che identifica il leader del cluster a cui appartiene.
- La lista dei vicini fisici `neighbors`.

2. **Nodi leader**: sono responsabili della gestione del cluster. Ogni nodo leader mantiene:
   - Le informazioni di se stesso in quanto nodo (`node`).
   - Il colore del cluster `color`.
   - L'ultimo evento `last_event` che il nodo ha gestito (i.e. cambio colore o merge) con il relativo timestamp.
   - La lista dei cluster adiancenti `adj_clusters`.
   - La lista `cluster_nodes` di tutti i nodi del cluster, compreso sè stesso.

Server condiviso

Il sistema utilizza un server condiviso per garantire la consistenza globale e sincronizzare le operazioni tra i leader dei cluster. Il server mantiene due strutture chiave: il **file di log** e lo **stato globale dei cluster**, entrambe cruciali per il corretto funzionamento e coordinamento del sistema.

1. **File di log**: un registro cronologico delle operazioni eseguite nel sistema, che include informazioni dettagliate su ogni azione eseguita. Ogni operazione nel file di log è composta dai seguenti elementi:
   - **Timestamp**: Il momento in cui l'operazione è stata eseguita, usato per ordinare temporalmente le azioni.
   - **Azione**: Il tipo di operazione effettuata, che può essere `merge` (unione di due cluster) oppure `color` (cambio di colore di un cluster).
   - **Colore**: il colore dell'operazione.
   - **From**: l'identificatore del leader del cluster in cui viene eseguita l'operazione.

```
{
  (timestamp, azione, colore, from),
  ...
}
```

Figura 3.2: Struttura del file di log

2. **Stato globale dei cluster**: Una matrice che rappresenta lo stato corrente di tutti i cluster nel sistema. Questa struttura consente al server di tenere traccia dello stato attuale e permette di gestire richieste e operazioni con efficienza. Per ogni leader, vengono mantenute le seguenti informazioni:
   - **Identificatore del leader** (`leaderID`): l'ID del leader del cluster.
   - **Nodi del cluster**: la lista dei nodi che appartengono al cluster.

- **Colore attuale del cluster** (`color`): Il colore corrente assegnato a tutti i nodi del cluster.
- **Cluster adiacenti**: La lista dei leader dei cluster adiacenti nel grafo overlay, che permette ai leader di comunicare tra loro e coordinare operazioni come il merge.

{
(leaderID, [nodo_1, nodo_2, ...], colore, [adj_cluster_1, adj_cluster_2, ...]),
...
}

Figura 3.3: Struttura dello stato globale dei cluster

3.2.4 Descrizione degli algoritmi

In questa sezione vengono descritti gli algoritmi chiave utilizzati nel sistema.

1. Algoritmi di scambio messaggi

In questa sezione descriviamo gli algoritmi utilizzati in ciascuna fase di scambio messaggi tra nodi interni e leader.

1.1 Inizializzazione nuova richiesta Ogni nodo ha vari tipi di richieste che può inoltrare al leader (3.2.5). Nel seguente pseudo-codice viene preso ad esempio il caso di richiesta di cambio colore da parte di un nodo (`change_color_request`).
Il nodo scelto inizializza una richiesta di cambio colore, creando un messaggio con tag `change_color_request` assieme all'evento contenente le informazioni necessarie per la richiesta. Il nodo inoltra la richiesta direttamente al leader.

Algorithm 1 Inizializzazione richiesta di cambio colore

```
1: procedure InitChangeColorRequest(newColor)
2:     event ← new('color', newColor, node.leaderID);
3:     tries ← 0;
4:     do
5:         response ← await send(node.leaderID, 'change_color_request', event);
6:         tries ← tries + 1
7:     while not response and tries <= 3
8: end procedure
```

1.2 Gestione richieste del leader Il leader del cluster può ricevere vari tipi di messaggi, ma nel seguente frammento di pseudo-codice viene posto il focus solo sulla gestione di richieste di cambio colore o di merge. Ciascuna richiesta viene prima inoltrata al server, il quale si occuperà di gestire casi particolari di concorrenza e di memorizzare l'operazione per la ridondanza. Una volta ricevuta risposta positiva dal server, il leader procede con l'esecuzione locale dell'operazione.

Algorithm 2 Gestione delle richieste nodo leader

```
1: procedure leaderReceiveRequest(event)
2:     {timestamp, type, color, from} ← event;
3:     if timestamp > leader.last_event.timestamp then
4:         response ← await send(server, event);
5:         if response == 'ok' then
6:             if type == 'color' then
7:                 changeColor(color);
8:             else if type == 'merge' then
9:                 merge(color);
10:            end if
11:        end if
12:    end if
13: end procedure
```

Il leader confronta il timestamp del messaggio con quello dell'ultima operazione eseguita. Se il timestamp del messaggio è più recente, il leader notifica il server dell'operazione ed in seguito alla ricezione dell'ok la esegue. La nuova operazione viene registrata nel file di log del server e lo stato globale del cluster viene aggiornato. Nel caso di un'operazione di cambio colore, il leader si occupa di controllare se eseguire un merge o meno in base al colore dei cluster adiacenti.

2. Algoritmo di cambio colore

Nell'operazione di cambio colore viene eseguito sia l'aggiornamento effettivo del parametro `color` del cluster, sia vengono gestite le altre operazioni di controllo ed aggiornamento. Viene infatti aggiornato il campo `last_event`, viene controllata la necessità di eseguire il merge ed infine viene inviata una richiesta (`color_adj_update`) di aggiornamento del colore ai cluster adiacenti per le loro liste `adj_clusters`. Alla fine dell'operazione viene comunicato al server la terminazione dell'algoritmo.

3. Algoritmo di merge

Il merge tra due (o più) cluster adiacenti viene eseguito quando il leader di un cluster effettua un cambio di colore e scopre che vi è almeno un cluster adiacente che condivide lo stesso colore. Il leader che ha avviato il cambio colore invia la richiesta di merge al leader del cluster adiacente con lo stesso colore.

3.2.5 Gestione dei messaggi e della comunicazione

La comunicazione tra i nodi è un aspetto fondamentale per il corretto funzionamento del sistema distribuito, specialmente in un contesto di sincronizzazione e consistenza. Di seguito vengono analizzati i tipi di messaggi utilizzati, il protocollo di comunicazione e la gestione di eventuali ritardi o fallimenti.

Tipi di messaggi

Nel sistema sono utilizzati diversi tipi di messaggi per coordinare le operazioni, sia tra i nodi normali che tra i leader e il server centrale:

- `get_leader_info`: utilizzato per ottenere le informazioni di un cluster dato l'ID del leader.
- `change_color_request`: inviato dai nodi normali verso il leader del proprio cluster quando desiderano avviare un'operazione di cambio colore. Questo messaggio viene inoltrato assieme a un record `event` contenente le informazioni sul nuovo colore e sul timestamp (vedi sezione 3.2.4).
- `update_adj_clusters`: notifica un leader di aggiornare la lista `adj_clusters` con i parametri passati.
- `merge_request`: messaggio inviato dal leader del cluster che ha appena cambiato colore verso i leader dei cluster adiacenti(vedi sezione 3.2.4).
- `aggiorna_leader`: aggiorna l'ID del leader in seguito ad un'operazione di merge tra due o più cluster.
- `update_cluster_nodes`: aggiorna la lista `cluster_nodes` inserendo i nodi dei cluster uniti tramite merge.
- `new_leader`/`transform_to_normal_node`: messaggi per eleggere a leader un nodo normale e viceversa.
- `save_to_db`: aggiorna stato attuale del cluster nel database del server.
- **Messaggi di conferma (ACK)**: utilizzati per garantire l'affidabilità del sistema, questi messaggi confermano la corretta ricezione delle richieste critiche. In caso di mancata risposta, viene ritentato l'invio (per una spiegazione più dettagliata 3.2.5).
- **Messaggi di `heartbeat`**: inviati periodicamente tra le varie componenti del sistema, servono per monitorare la disponibilità e la salute dei nodi (normali e leader), e rilevare eventuali fallimenti (per una spiegazione dettagliata 3.2.8).
- **Messaggi al server**: i nodi leader comunicano con il server per aggiornare lo stato globale del cluster, registrare le operazioni nel file di log, e segnalare eventuali fusioni, cambi di colore o fallimenti.

Protocollo di comunicazione

Per garantire un alto grado di affidabilità e tolleranza ai guasti, il sistema implementa un protocollo basato su conferme (ACK) e ritrasmissioni in caso di timeout:

- **Invio del messaggio**: ogni volta che un nodo invia un messaggio critico (i.e. `change_color_request` o `merge_request`), avvia un timer di timeout.
- **Conferma di ricezione (ACK)**: il nodo ricevente, dopo aver elaborato il messaggio, invia un `ACK` al mittente per confermare la corretta ricezione e processazione.
- **Ritrasmissione in caso di timeout**: se il mittente non riceve l'`ACK` entro un tempo predefinito, ritrasmette il messaggio. Viene eseguito un numero massimo di ritrasmissioni prima che il nodo consideri il destinatario come non raggiungibile.
- **Numero massimo di tentativi**: se un nodo non riceve conferma dopo un certo numero di tentativi, considera il nodo destinatario come fallito e inizia le procedure di recupero.

Gestione dei ritardi e dei messaggi fuori ordine

Considerando che i messaggi in rete possono subire ritardi e arrivare fuori ordine, il sistema utilizza meccanismi di controllo dei timestamp e gestione delle operazioni per evitare incongruenze:

- **Timestamp e identificatori univoci**: ogni messaggio è accompagnato da un timestamp e un identificatore univoco per garantire che sia elaborato correttamente e nell'ordine appropriato. I nodi utilizzano questi dati per determinare l'ordine delle operazioni.
- **Gestione dei messaggi duplicati**: il sistema ignora automaticamente i messaggi duplicati (ossia già ricevuti ed elaborati), riducendo l'overhead di calcolo e prevenendo errori di consistenza.
- **Ordinamento delle operazioni**: nel caso di ricezione di messaggi fuori ordine, il sistema applica le operazioni basandosi sui timestamp per garantire che vengano eseguite nel corretto ordine temporale.

3.2.6 Gestione dei timestamp

Per garantire un ordinamento coerente degli eventi e mantenere la consistenza causale, è stato scelto di utilizzare un processo a sè stante. Nuovi eventi vengono creati unicamente da questo processo, garantendo che così che il clock con cui vengono generati i timestamp sia lo stesso, indipendentemente dal quale sia il nodo per cui è stato creato l'evento. A differenza di un approccio in cui ogni processo nodo ha il proprio separato clock, l'impiego di un processo separato per gli eventi evita sia possibili casi di asincronia dei clock, sia l'overhead dovuto alla regolare sincronizzazione dei clock.
L'impiego dei timestamp ed id risulta fondamentale per la gestione sia delle richieste di cambio colore che di quelle di merge, poiché, per risolvere casi di concorrenza, viene fatto un ordinamento delle operazioni basato in primis sul tempo di invio della richiesta e secondariamente sull'identificativo dell'operazione, nei particolari casi in cui il tempo sia precisamente lo stesso. L'ordinamento permette sia di gestire in maniera precisa le possibili situazioni di concorrenza affrontate nella sezione 3.2.6, sia di registrare le operazioni in un unico file di log.

Utilizzo nel sistema Il sistema prevede che ad ogni nuova richiesta di cambio colore o merge, venga contattato il processo degli eventi e questo risponda con un nuovo record con i parametri forniti.

Risoluzione dei conflitti In caso di conflitti tra operazioni concorrenti (ad esempio, due richieste ricevute quasi simultaneamente), l'operazione con il timestamp minore ha la precedenza. In caso di parità di timestamp, viene utilizzato l'id dell'evento.

Considerazioni sulla scelta È stato preso in considerazione anche l'utilizzo di un orologio logico, in particolare quello proposto da Lamport. Tuttavia, data sia la struttura di rete, sia la direzione dei messaggi trasmessi (convergente verso il leader), l'utilizzo di timestamp generati da un processo centrale risulta più efficiente in termini di performance e numero di messaggi scambiati. L'implementazione dell'orologio logico di Lamport richiederebbe un continuo scambio di messaggi tra i nodi per mantenere i timestamp locali sincronizzati, con un overhead significativo. Al contrario, nella soluzione scelta non vi è necessità di meccanismi di sincronizzazione e lo scambio di messaggi è limitato all'indispensabile: ad ogni nuova richiesta di operazione da un nodo, vengono trasmessi nella rete solo 2 messaggi, ovvero la richiesta di un nuovo record evento e la risposta contenente esso.
Come descritto nel dettaglio nella precedente sezione 3.2.6 sulla comunicazione, ogni richiesta confluisce verso il leader e il sistema di request-response basato su ACK è implementato unicamente tra due nodi direttamente comunicanti. L'utilizzo dell'orologio logico di Lamport, senza l'aggiunta di messaggi di risposta in direzione contraria, causerebbe col tempo un disallineamento tra i timestamp dei nodi meno attivi e quelli del leader, rendendo le loro richieste sempre più obsolete e infine scartate.
Inoltre, l'orologio logico di Lamport risulterebbe problematico per un ordinamento globale delle operazioni all'interno del log, poiché ogni cluster sarebbe temporalmente indipendente dagli altri, legato al numero di richieste ricevute dal proprio leader.

Limitazioni dell'approccio Nonostante i vantaggi, questo approccio presenta la criticità d'introdurre un singolo punto di fallimento nel sistema: il processo *event*. Nel caso in cui dovesse fallire o diventare non raggiungibile, non sarebbe più possibile creare eventi e tutte le nuove richieste di operazioni resterebbero in attesa. Nel caso in cui non fosse garantita la

continuità del servizio, all'evenienza del seguente problema, l'intero sistema sarebbe compromesso e resterebbe in uno stato di stand-by. Per questo è stato deciso di monitorare il processo event, permettendo il riavvio istantaneo del processo, e sono state implementate delle code di messaggi nei nodi, permettendo di accumulare nuove richieste di operazioni, evitando perdite durante il riavvio del processo event e garantendo il ripristino allo stato esatto al momento del guasto.

3.2.7 Gestione della Consistenza

Per garantire il corretto funzionamento del sistema e preservare la coerenza degli stati tra i vari cluster, vengono gestiti i seguenti possibili casi di inconsistenza:

Caso 1: Richiesta di Cambio Colore con Timestamp Anteriore che Richiede un Merge Quando, dopo un'operazione di cambio colore con timestamp T_2, arriva una richiesta con un timestamp precedente $T_1 < T_2$ che avrebbe comportato un merge con un cluster adiacente al tempo T_1 (Figura 3.4):

- Si riportano le operazioni di ripristino per tornare allo stato registrato al tempo T_1.
- Si procede quindi con il merge tra i due cluster.
- Infine, si applica il nuovo colore risultante dall'operazione più recente.

Questo assicura che il merge mancato al tempo T_1 venga comunque eseguito, integrando il nuovo stato generato dalle operazioni più recenti.

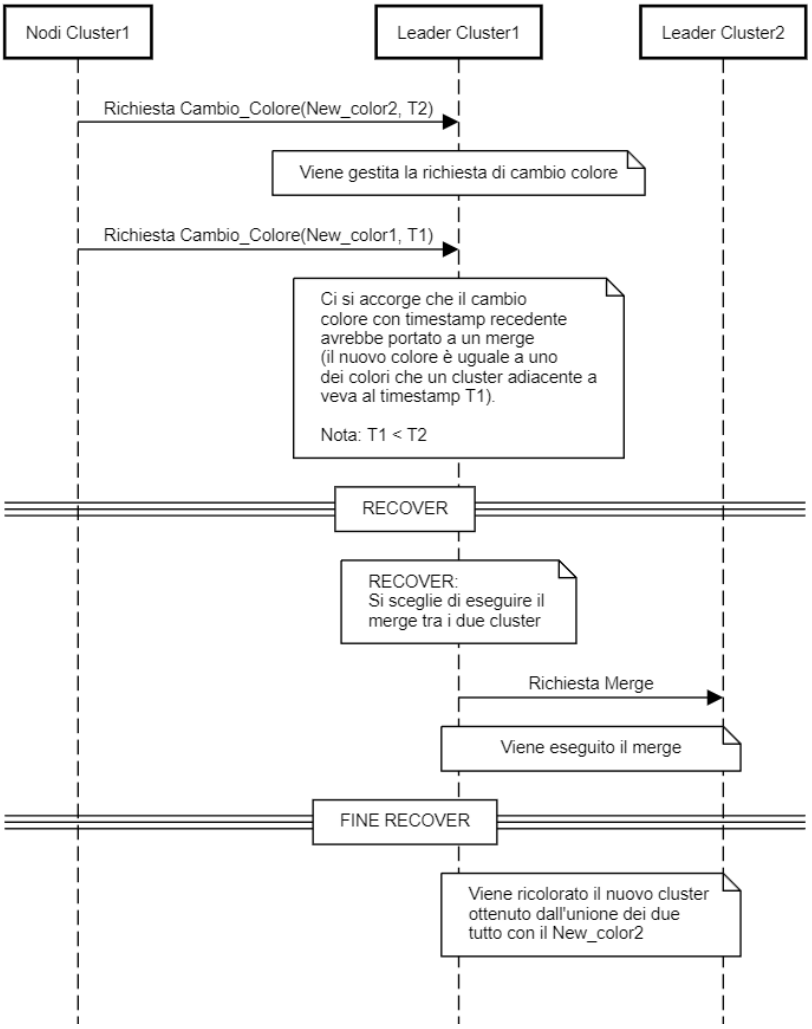


Figura 3.4: Caso 1: Richiesta di cambio colore con timestamp anteriore che richiede un merge

Caso 2: Richiesta di Cambio Colore con Timestamp Anteriore che Non Richiede un Merge Nel caso in cui una richiesta con timestamp $T_1 < T_2$ non avrebbe comportato un merge al tempo T_1, ma l'operazione successiva con T_2 ha già effettuato un merge (Figura 3.5):

- La richiesta con timestamp T_1 viene ignorata, poiché lo stato corrente non rispecchia più le condizioni esistenti al tempo T_1.

Questo evita di sovrascrivere cambiamenti significativi con operazioni ormai obsolete.

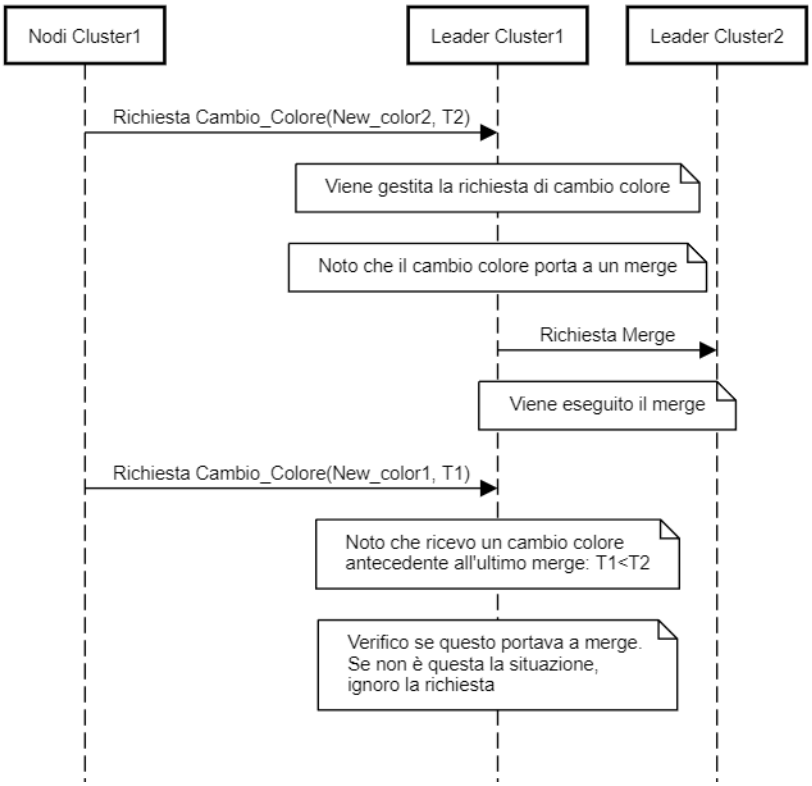


Figura 3.5: Caso 2: Richiesta di cambio colore con timestamp anteriore che non richiede un merge

Caso 3: Richiesta di Merge Ricevuta durante un Cambio Colore con Timestamp Anteriore Quando un leader riceve una richiesta di merge mentre è in corso un'operazione di cambio colore con un timestamp anteriore (Figura 3.6):

- Il leader attende per un periodo di tempo prestabilito per consentire l'arrivo di eventuali richieste tardive.
- Se dopo il timeout non giungono ulteriori richieste, il leader procede con l'operazione di merge.

Questo meccanismo previene eventuali inconsistenze dovute all'ordine fuori sincronia delle operazioni.

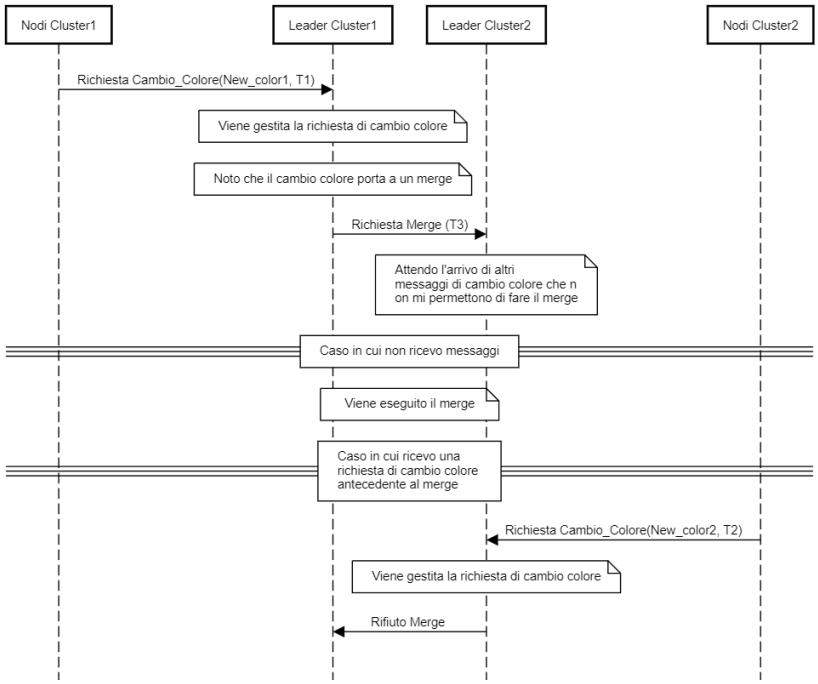


Figura 3.6: Caso 3: Richiesta di merge ricevuta mentre è in corso un cambio colore con timestamp anteriore

Caso "Too Old": Gestione di una Richiesta con Timestamp Troppo Vecchio
Quando un leader riceve una richiesta di cambio colore o di merge con un timestamp troppo vecchio rispetto all'ultima operazione completata (Figura 3.7):

- La richiesta viene ignorata ("droppata") poiché il suo stato è ormai obsoleto rispetto alle operazioni più recenti.

Questo assicura che non vengano applicate modifiche non più pertinenti, evitando di introdurre inconsistenze nel sistema.

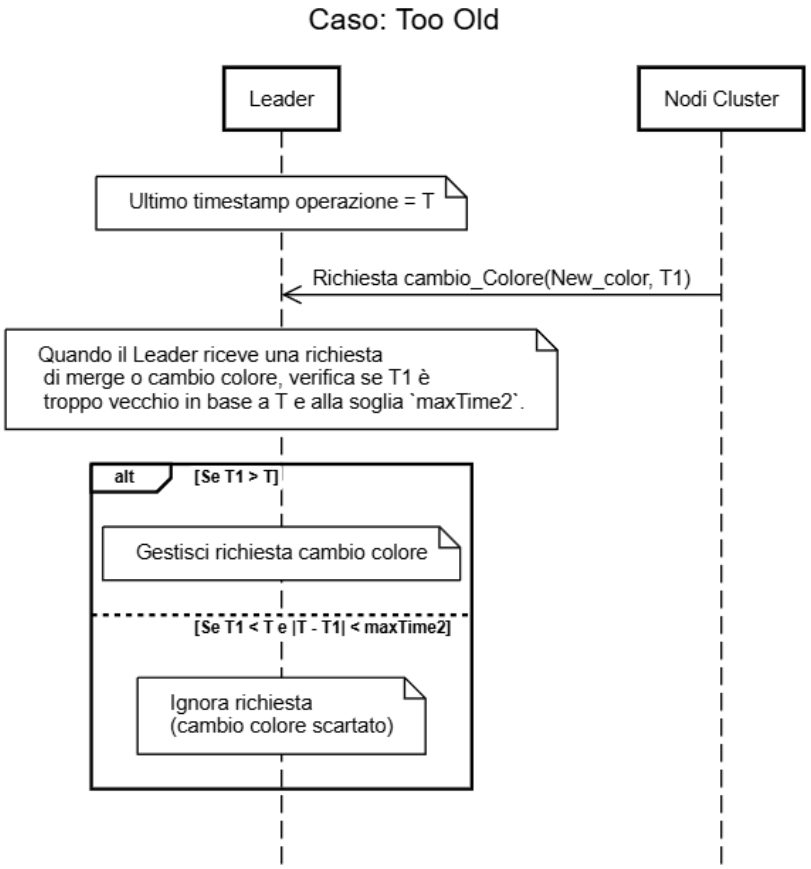


Figura 3.7: Caso "Too Old": Richiesta con timestamp troppo vecchio

Caso "Cambio Colore durante Merge": Richiesta di Cambio Colore Ricevuta durante un Merge Se un leader riceve una richiesta di cambio colore o di merge mentre è in corso un'operazione di merge (Figura 3.8):

- La richiesta viene accodata e verrà eseguita una volta completato il merge in corso.

In questo modo, si evita di alterare lo stato corrente mentre è in corso il merge, garantendo la coerenza e l'ordine delle operazioni.

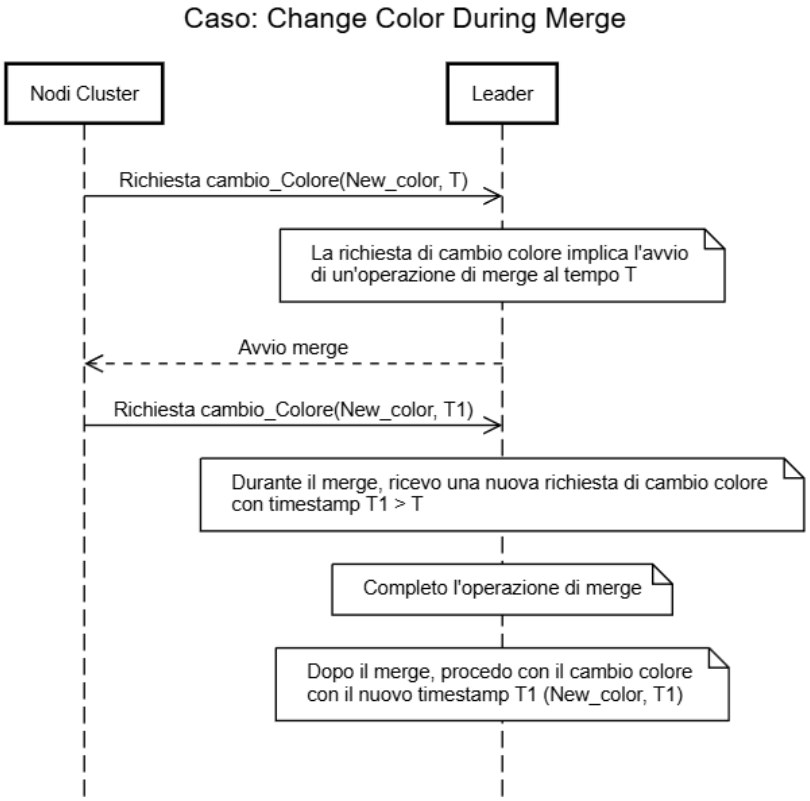


Figura 3.8: Caso "Cambio Colore durante Merge": Richiesta di cambio colore durante un merge

Caso "Doppio Merge": Merge Tra Cluster Adiacenti con Cambio Colore Nel caso in cui due cluster adiacenti, inizialmente di colori diversi, ricevano una richiesta di cambio colore che li porta entrambi ad assumere lo stesso colore (Figura 3.9):

- Si esegue un'operazione di merge, unendo i due cluster in un unico grande cluster con il nuovo colore comune.

Questo approccio garantisce che il sistema mantenga uno stato consistente e riduce la frammentazione dei cluster.

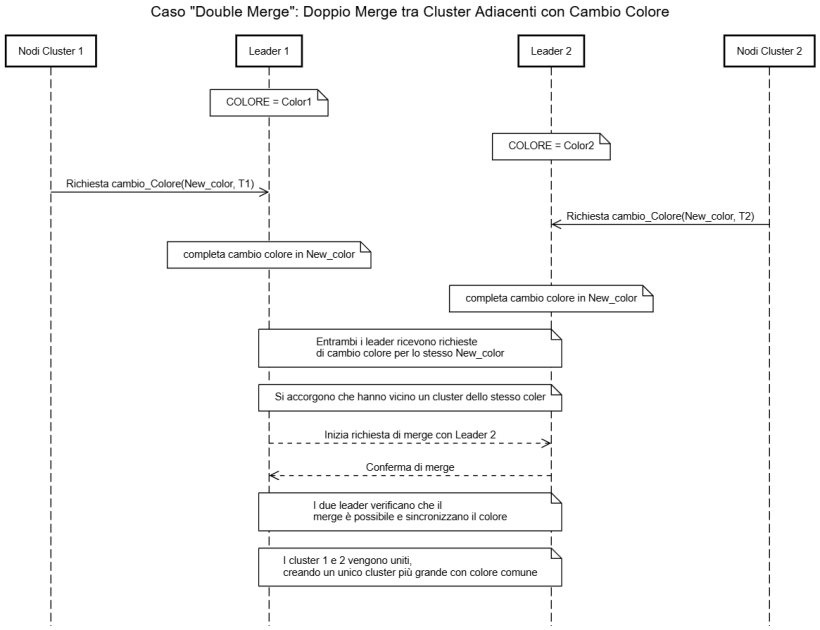


Figura 3.9: Caso "Doppio Merge": Merge tra cluster adiacenti con cambio colore

3.2.8 Gestione dei fallimenti

La robustezza del sistema dipende dalla capacità di gestire efficacemente i fallimenti dei nodi, sia leader che normali. Di seguito vengono dettagliate le strategie adottate per ricalcolare la struttura del cluster in caso di fallimenti, con particolare attenzione ai casi limite.

1. Fallimento di un nodo leader

Il leader di un cluster è fondamentale per la coordinazione e la comunicazione all'interno del cluster stesso. Tuttavia, il fallimento di un leader può generare scenari complessi, come la formazione di sotto-grafi disconnessi. Il sistema deve quindi gestire efficacemente tali situazioni per garantire la continuità operativa e la coerenza del cluster.

- **Monitoraggio tramite heartbeat:** I leader e il server scambiano continuamente messaggi di `heartbeat` per monitorare lo stato di salute dei leader.
- **Rilevamento del fallimento:** Se il server o un altro leader non riceve un `heartbeat` da un leader entro un tempo prefissato (*timeout*), assume che il leader sia fallito.
- **Gestione del partizionamento in sotto-grafi:**

  - Quando un leader fallisce, il server elegge il nodo con l'`id` più basso rimasto e lo nomina come nuovo leader.
  - Il nuovo leader avvia una BFS per ristabilire la struttura gerarchica del suo sotto-grafo e comunica al server i nodi che fanno parte del sotto-grafo rimanente.
  - Il server controlla se ci sono nodi non inclusi nel nuovo sotto-grafo e, per ogni gruppo di nodi sconnessi, elegge altri nuovi leader per gestire i restanti sotto-grafi.
  - Il processo continua finché tutti i nodi disconnessi non vengono riassegnati a un leader e il cluster non è completamente ripristinato.

- **Ricostruzione della struttura del cluster:**
  - Durante la BFS, ogni nodo del sotto-grafo riassegnato aggiorna il proprio `leaderID` con quello del nuovo leader.
  - Vengono sistemate le relazioni di `parent` e `child` tra i nodi per garantire la connessione all'interno del sotto-grafo.
  - Vengono comunicati eventuali cluster adiacenti al nuovo leader in modo che possa aggiornare la rete overlay.

Casi limite da gestire:

- *Partizionamento in più sotto-grafi disconnessi:* Se il fallimento del leader causa la formazione di più sotto-grafi disconnessi, il server si assicura che ogni sotto-grafo venga gestito in modo indipendente con un nuovo leader. Il processo continua fino a quando tutti i nodi del cluster originario non sono stati riassegnati.
- *Inconsistenze durante l'elezione:* Per evitare condizioni di competizione (*race conditions*), i nodi seguono una politica deterministica basata sull'`id` più basso per eleggere il nuovo leader in ciascun sotto-grafo.

2. Fallimento di un nodo normale

Il fallimento di un nodo non leader può avere impatti diversi sulla struttura del cluster, a seconda del ruolo del nodo e della sua posizione all'interno dell'albero di comunicazione.

- **Nodo foglia:** Se il nodo fallito è una foglia, il suo fallimento non interrompe la comunicazione tra gli altri nodi del cluster. Nessuna azione è necessaria.
- **Nodo interno:** Se il nodo è interno e il suo fallimento interrompe la comunicazione tra parti del cluster, è necessario ricalcolare la struttura per ristabilire la connettività. Questo processo coinvolge diverse fasi dettagliate di seguito.

1. **Rilevamento del fallimento:**
   - **Tentativi di comunicazione:** Il nodo figlio del nodo fallito tenta di comunicare con il proprio `parent` per operazioni periodiche (ad esempio, messaggi di `heartbeat`) o in risposta a eventi specifici (come richieste di cambio colore).
   - **Timeout e assunzione di fallimento:** Se il nodo figlio non riceve risposta dopo un certo numero di tentativi, utilizzando meccanismi di `ack` (acknowledgment) e un intervallo di *timeout*, assume che il `parent` sia fallito.
2. **Inizio della prima BFS (riconfigurazione):**
   - **Avvio di una BFS locale:** Il nodo figlio avvia una *Breadth-First Search* (BFS) per esplorare i nodi raggiungibili con lo stesso `leaderID`, inviando messaggi di `BFS_Request` ai propri vicini. L'obiettivo è capire se esiste un percorso alternativo per raggiungere il leader originale.
   - **Tracciamento delle richieste:** Durante la BFS, ogni nodo che riceve un `BFS_Request` si segna da quale nodo è arrivata la richiesta, così da poter ripercorrere all'indietro il percorso seguito dalla richiesta una volta trovato un nodo connesso al leader.
3. **Analisi dei risultati della prima BFS:**
   (a) **Caso 1 - Cluster non isolato (leader raggiungibile):** Se la BFS raggiunge il leader originale il percorso seguito dalla prima BFS viene ripercorso all'indietro, aggiornando solo i nodi coinvolti direttamente nella ricostruzione del percorso e ristabilendo la connessione al nodo sconnesso aggiornando i `parent` dei nodi.
   (b) **Caso 2 - Cluster isolato (leader non raggiungibile):** Se la BFS non trova alcun percorso per raggiungere il leader originale, significa che il sotto-grafo è diventato isolato dal cluster principale. Il nodo che ha avviato la BFS attende prima di avviare altre azioni:
      - **Attesa delle risposte:** Il nodo attende di ricevere risposte da tutti i nodi raggiunti dalla BFS. Se nessuno dei nodi risponde di essere collegato al leader, il nodo si autoproclama leader.
      - **Avvio della seconda BFS (nuovo leader):** Una volta autoproclamato leader, il nodo avvia una seconda BFS per informare tutti i nodi del sotto-grafo che è diventato il nuovo leader.
      - **Aggiornamento dei nodi:** Tutti i nodi raggiunti dalla BFS aggiornano il proprio `leaderID` con quello del nuovo leader.
      - **Comunicazione al server:** Il nuovo leader invia un messaggio di `NewLeaderAnnouncement` al server, informandolo della formazione del nuovo cluster e fornendo la lista dei nodi coinvolti.

3.2.9 Criteri di selezione del leader

In un sistema distribuito come quello descritto, la selezione di un leader all'interno di un cluster di nodi è fondamentale per garantire efficienza operativa e tolleranza ai guasti. Nel contesto dell'algoritmo Flood-Fill distribuito, dove ogni nodo rappresenta un pixel, i leader devono essere scelti in modo da minimizzare la complessità e garantire la robustezza. Di seguito, vengono discussi i criteri utilizzati per la selezione del leader.

Attribuzione dell'ID ai nodi

Ogni nodo nel sistema rappresenta un pixel di una matrice bidimensionale, e a ciascun nodo viene assegnato un identificatore unico (ID). Per generare l'ID dei nodi, si utilizzano le coordinate del pixel `(x, y)` all'interno della matrice. Esistono diversi metodi per attribuire un ID:

- **Utilizzo delle coordinate `(x, y)`**: le coordinate stesse possono essere utilizzate come identificatore. In questo caso, l'ID del nodo sarà la coppia `(x, y)`, che rappresenta la posizione esatta del nodo nella matrice.
- **Linearizzazione delle coordinate**: per ottenere un ID scalare unico, è possibile convertire le coordinate `(x, y)` in un unico valore intero utilizzando la seguente formula:

  $$ID_{nodo} = y \times larghezza + x$$

  dove $larghezza$ è il numero di colonne della matrice. Questa rappresentazione permette di gestire più facilmente i nodi come valori interi unici.
- **Utilizzo dell'id del processo**: nello scenario di un sistema distribuito come questo, ogni nodo viene gestito come un processo a sè stante ed è possibile utilizzare l'id del processo come identificatore del nodo stesso.

Nel sistema proposto è stato scelto di utilizzare la terza soluzione, in quanto si adatta perfettamente al tipo di sistema. É stato inoltre scelto di sfruttare l'ideologia della prima proposta, creando dei riferimenti testuali `node_X_Y` per ogni PID[2], per facilitare il debugging degli scambi di messaggi e la lettura dei file di log e della struttura dati.

Selezione del leader

Una volta che a ciascun nodo è stato attribuito un ID, è necessario un criterio per la selezione del leader all'interno di ciascun cluster. I criteri di selezione possono variare in base alle esigenze del sistema, ma nel contesto di questo progetto si utilizza il seguente approccio:

ID più basso come leader Il criterio principale adottato è la selezione del nodo con l'ID più basso all'interno del cluster. Questo approccio offre diversi vantaggi:

- **Semplicità**: La selezione del leader basata sull'ID più basso è semplice da implementare e non richiede calcoli complessi o monitoraggi continui.
- **Determinismo**: Il nodo con l'ID più basso è sempre determinabile, garantendo che la selezione del leader sia univoca e ripetibile.

Failover e rielezione Nel caso in cui il leader selezionato fallisca, è previsto un meccanismo di failover automatico. I nodi del cluster avviano un'elezione per scegliere un nuovo leader, basata nuovamente sul criterio dell'ID più basso tra i nodi rimanenti. Questo processo garantisce che il cluster possa continuare a funzionare correttamente anche in presenza di guasti.

[2]Process ID

3.2.10 Vantaggi della soluzione proposta

La soluzione scelta presenta diversi vantaggi:

- **Scalabilità**: trattando i cluster come nodi in un grafo overlay, il sistema può gestire efficacemente reti di grandi dimensioni.
- **Consistenza globale**: l'uso del database condiviso garantisce la sincronizzazione delle operazioni e la consistenza dello stato globale.
- **Robustezza ai fallimenti**: i meccanismi di gestione dei fallimenti assicurano che il sistema possa recuperare rapidamente da guasti di nodi leader o interni.
- **Efficienza**: la comunicazione è ottimizzata utilizzando i nodi leader all'interno dei cluster e il grafo overlay tra i leader.

3.3 Piano di Sviluppo

Il piano di sviluppo del sistema è organizzato in sei fasi principali, ciascuna mirata a implementare aspetti cruciali del sistema distribuito, iniziando dalle funzionalità di base fino ad arrivare alla gestione di consistenza e fault tolerance. Di seguito, vengono dettagliate le fasi del progetto:

Fase 1: Implementazione della Comunicazione Base e Setup

La prima fase è focalizzata sull'implementazione della comunicazione essenziale tra i nodi e sulla corretta configurazione della rete. L'obiettivo è garantire che ogni nodo possa scambiare messaggi con i nodi vicini e che la fase di setup, inclusa l'assegnazione dei leader e la costruzione del grafo dei cluster, funzioni correttamente.

Fase 2: Implementazione dell'Algoritmo di Cambio Colore

Una volta che la comunicazione base è stabilita, viene implementato l'algoritmo di cambio colore. Questo algoritmo permetterà a un nodo di avviare l'operazione di ricolorazione che si propagherà ai nodi adiacenti dello stesso cluster, sotto il coordinamento del leader.

Fase 3: Implementazione dell'Algoritmo di Merge

Nella terza fase, viene sviluppato l'algoritmo di merge, che gestisce la fusione di cluster adiacenti quando questi acquisiscono lo stesso colore. L'algoritmo deve garantire che i leader dei cluster coinvolti collaborino per eseguire l'unione in maniera efficiente e consistente.

Fase 4: Implementazione del Server di Gestione del Tempo e Server Centrale

In questa fase, viene implementato un server centrale che gestisce i timestamp e coordina le operazioni tra i cluster. Il server avrà il compito di mantenere una visione globale dello stato del sistema e garantire che gli eventi vengano ordinati correttamente nel tempo, migliorando così la coerenza globale.

Fase 5: Gestione della Consistenza Globale

Una volta implementato il server centrale, viene sviluppato un sistema di gestione della consistenza globale. Questo sistema garantisce che le operazioni distribuite, come i cambi di colore e i merge, siano ordinate in maniera corretta, evitando conflitti e garantendo che tutti i nodi abbiano una visione coerente dello stato del sistema.

Fase 6: Gestione dei Fallimenti

Nell'ultima fase, vengono implementati i meccanismi per gestire i fallimenti dei nodi, sia normali che leader. Questo include la rilevazione dei fallimenti, l'elezione di nuovi leader in caso di guasti e la riconfigurazione del sistema per mantenere la continuità operativa anche in presenza di nodi falliti. L'obiettivo è garantire che il sistema sia resiliente e possa continuare a funzionare correttamente anche in condizioni di fault.

3.4 Conclusioni

La soluzione proposta, basata sul trattamento dei cluster come nodi in un grafo overlay e sull'uso di un database condiviso per la sincronizzazione, offre un equilibrio tra efficienza, scalabilità e robustezza. I meccanismi di gestione della consistenza e dei fallimenti garantiscono il corretto funzionamento del sistema in ambienti distribuiti e soggetti a guasti.

Capitolo 4

Implementazione

4.1 Scelta del linguaggio di programmazione

Per questo progetto distribuito sono stati utilizzati due linguaggi di programmazione principali: **Erlang** e **Python**. Ciascuno di essi è stato scelto per le sue caratteristiche specifiche, che permettono di risolvere efficacemente problemi di distribuzione e visualizzazione.

4.1.1 Erlang

Erlang è stato scelto per la logica distribuita e la gestione dei nodi nel sistema, grazie alla sua robustezza e al supporto nativo per la concorrenza e la tolleranza ai guasti. Erlang è particolarmente adatto per applicazioni distribuite e fault-tolerant, ed è ampiamente utilizzato in ambiti come le telecomunicazioni e i sistemi di backend ad alta disponibilità.

- **Gestione dei processi**: in Erlang, ogni nodo e leader è gestito come un processo indipendente. Questo permette al sistema di scalare facilmente e di mantenere l'isolamento dei fallimenti tra i nodi.
- **Pattern matching e concorrenza**: Erlang utilizza il pattern matching per la gestione dei messaggi e le strutture dati, facilitando la scrittura di codice conciso e mantenibile. La concorrenza è una funzionalità nativa del linguaggio, che permette ai processi di comunicare tramite messaggi in maniera asincrona.
- **Semplicità e resilienza**: Erlang implementa una semplice struttura di error handling in cui, invece di cercare di prevenire ogni possibile errore, i processi possono fallire e riavviarsi autonomamente. Questo modello "let it crash" è efficace in sistemi distribuiti dove i singoli componenti devono rimanere resilienti.

Erlang è quindi stato impiegato per la creazione e gestione dei nodi, la comunicazione tra essi e per mantenere il server centrale che coordina i leader e gestisce la propagazione dei messaggi.

4.1.2 Python

Python è stato scelto per la visualizzazione dei dati e l'interfaccia utente. Con la libreria `Flask`, è stato possibile implementare un server web per mostrare la matrice dei nodi e permettere l'interazione dell'utente, mentre `Matplotlib` è stato utilizzato per la grafica della griglia.

- **Visualizzazione e interfaccia web**: Python, con `Flask` e `Matplotlib`, permette di creare interfacce web e rappresentazioni grafiche dei nodi in modo rapido e flessibile.

- **Aggiornamenti in tempo reale**: Il supporto di `Flask-SocketIO` consente la comunicazione in tempo reale tra il backend e l'interfaccia utente, permettendo di aggiornare la visualizzazione della griglia dei nodi ogni volta che avvengono modifiche nel sistema.
- **Facilità d'integrazione con Erlang**: Python comunica con Erlang tramite un server TCP implementato in Erlang stesso. Questo approccio rende semplice l'integrazione e il passaggio di messaggi per operazioni come il cambio di colore dei nodi.

Python è quindi stato utilizzato come componente di frontend, fornendo una rappresentazione visiva intuitiva e permettendo l'interazione con i nodi per modificarne lo stato in tempo reale.

4.1.3 Benefici della combinazione di Erlang e Python

La combinazione di Erlang e Python ha permesso di sfruttare il meglio di entrambi i linguaggi:

- **Distribuzione e resilienza** con Erlang per la gestione dei nodi e della logica di sistema.
- **Visualizzazione intuitiva e interattiva** con Python per fornire un'interfaccia utente che permette di osservare lo stato del sistema e interagire con esso.

Questo approccio ha consentito di mantenere un sistema distribuito robusto e allo stesso tempo di offrire una visualizzazione efficace e flessibile.

4.2 Comandi principali per l'avvio del sistema

In questa sezione sono descritti i comandi principali per avviare e configurare il sistema distribuito Flood-Fill. È possibile eseguire il sistema utilizzando due o tre shell separate, a seconda delle esigenze: - **Con due shell**: ideale per un uso manuale, dove una shell avvia il backend Erlang e l'altra avvia il server Flask per l'interfaccia web. - **Con tre shell**: utile per test automatizzati, aggiungendo una shell dedicata agli script di testing, che inviano richieste al sistema per verificare la correttezza e la resilienza in tempo reale.
I comandi descritti possono essere eseguiti sia manualmente per monitorare il sistema in modo interattivo, sia con script automatici di testing per simulare carichi di lavoro o condizioni specifiche.

4.2.1 Avvio del Backend Erlang

Questo comando compila e avvia il backend, configurando una griglia di nodi e assegnando una porta per la comunicazione.

Comandi di Avvio in Erlang

```
./compile_and_run.sh <ROWS> <COLUMNS> <FROM_FILE>
```

Dove:

- `<ROWS> <COLUMNS>` sono le dimensioni della griglia (ad esempio, 7 7 per una griglia 7x7).
- `<FROM_FILE>` specifica se i colori dei nodi devono essere letti da un file (`True`) o generati casualmente (`False`).

4.2.2 Avvio del Server Flask

Il server Flask fornisce l'interfaccia web, che permette agli utenti di interagire con la griglia e cambiare i colori dei nodi.

Comandi di Avvio del Server Flask

```
python3 grid_visualizer.py --debug <DEBUG_MODE> --port <PORTA>
```

Dove:

- `<DEBUG_MODE>` è `True` per attivare la modalità di debug, utile durante la fase di sviluppo.
- `<PORTA>` è la porta fornita dal backend Erlang per la comunicazione.

4.2.3 Generazione di Colori e Operazioni Casuali

Questo comando genera una griglia di nodi con colori casuali e una lista di operazioni di cambio colore preimpostate, che possono essere usate per simulare vari scenari di test.

Generazione di Colori e Operazioni Casuali

```
python3 generate_changes_rand.py --rows <ROWS> --columns <COLUMNS>
                                 --operations  <NUMBER_OPERATIONS>
```

Dove:

- `<NUMBER_OPERATIONS>` è il numero di cambi di colore casuali da generare.

4.2.4 Esecuzione di Test Automatici

Lo script di test automatico invia una serie di richieste di cambio colore al backend per verificare che i cambiamenti siano correttamente propagati e gestiti dal sistema.

Esecuzione di Test Automatici

```
python3 script.py
```

Questi test permettono di osservare il comportamento del sistema in condizioni di carico e di verificare la consistenza delle operazioni di cambio colore e la resilienza del sistema in caso di errori.

4.2.5 Esecuzione in Shell Multiple

Per eseguire il sistema con **due shell**, avvia il backend Erlang nella prima shell e il server Flask nella seconda. Questa configurazione è adatta per un uso manuale, poiché consente di monitorare i log e l'interfaccia web direttamente.
Per eseguire il sistema con **tre shell**, usa una terza shell per avviare script di test automatici. Questo approccio è utile per verificare la robustezza del sistema, poiché permette di generare condizioni di carico e testare la propagazione dei cambi di colore tra i nodi.

4.3 Records

Le principali strutture dati utilizzate sono rappresentate dai record 'node' e 'leader', che organizzano le informazioni di ciascun nodo e dei leader dei cluster. Questi record sono definiti in `node.erl` e permettono di gestire la posizione, i vicini e l'ID del leader per ciascun nodo.

4.3.1 Record `node.hrl`

Il record `node` rappresenta ciascun nodo del sistema. Contiene campi per le coordinate, il nodo parent, i figli, e lo stato di visita.

Definizione del record `node`

```
-record(node, {pid, x, y, leaderID, neighbors = []}).
```

- **pid**: identificatore del processo Erlang associato al nodo.
- **x, y**: coordinate del nodo nella griglia.
- **leaderID**: pid del leader del cluster a cui il nodo appartiene.
- **neighbors**: lista dei pid dei nodi fisicamente adiacenti.

4.3.2 Record `leader.hrl`

Il record `leader` rappresenta i leader dei cluster, gestendo l'aggregazione di nodi con lo stesso colore. Questo record incapsula il record nodo, facilitando il passaggio da nodo normale a leader e viceversa. Ogni leader contiene informazioni sia sul proprio cluster che di quelle dei cluster adiacenti.

Definizione del record `leader`

```
-record(leader, {
    node, color, last\_event, adj\_clusters = [], cluster\_nodes = []
  }).
```

- **node**: struttura dati del nodo leader.
- **color**: colore del cluster.
- **last_event**: informazioni sull'ultima operazione eseguita sul cluster.
- **adj_clusters**: lista dei cluster adiacenti.
- **cluster_nodes**: lista dei pid dei nodi appartenenti al cluster.

4.3.3 Record `event.hrl`

Il record `event` viene usato per date una struttura alle informazioni sulle operazioni che vengono eseguite sui cluster.

Definizione del record `event`

```
-record(event, {timestamp, id, type, color, from}).
```

- **timestamp**: tempo al momento della creazione di una nuova richiesta (operazione di cambio colore o merge).
- **id**: identificatore univoco della richiesta. Essendo progressivi, nel caso di concorrenza (i.e. timestamp equivalente) viene usato per determinare l'ordine.
- **type**: tipo di operazione (`color` o `merge`).
- **color**: colore associato all'operazione.
- **from**: leaderID del cluster in cui avverà l'operazione. Usato dal server per conoscere il richiedente dell'operazione.

4.4 Moduli

4.4.1 Modulo `node.erl`

Il modulo `node.erl` è progettato per gestire la creazione, la gestione e la comunicazione tra i nodi in una rete distribuita. Esso permette di coordinare dinamicamente le operazioni sui nodi e sui leader all'interno del sistema, gestendo anche eventi come cambi di colore, elezione di nuovi leader e fusioni di cluster.
Il modulo contiene le seguenti funzioni principali:

- `new_node/8`: Questa funzione crea un nuovo nodo con parametri specifici come la posizione, il leader, i vicini e altre proprietà essenziali. Ogni nodo è dotato di un identificativo unico (PID) e può interagire con i vicini e il leader assegnato.
- `new_leader/5`: Crea e configura un nodo come leader. Un leader coordina i nodi all'interno di un cluster, assegnandosi il proprio PID come leader ID. La funzione inizializza inoltre attributi specifici come il colore del cluster e le informazioni sui cluster adiacenti.
- `create_node/2`: Avvia il processo per un nodo leader, aggiornando il suo PID e garantendo che possa comunicare con gli altri nodi della rete. Questa funzione spawna un processo per il ciclo di vita del nodo, rendendolo operativo.
- `leader_loop/1`: È il ciclo principale del leader. Questa funzione permette al leader di ricevere e gestire continuamente i messaggi, garantendo una risposta reattiva agli eventi che si verificano nella rete, come aggiornamenti di cluster o cambiamenti di colore.
- `node_loop/1`: Ciclo principale per i nodi standard, che processa i messaggi ricevuti e permette al nodo di rispondere a richieste di cambio colore, aggiornamenti di leader e altre interazioni con i nodi vicini.

Lo scopo del modulo `node.erl` è fornire una struttura robusta per la gestione di una rete distribuita di nodi, dove ciascun nodo può essere configurato come nodo standard o leader. La flessibilità del modulo consente di implementare dinamicamente una rete in cui i nodi possono cambiare colore, coordinarsi con i leader e gestire le proprie interazioni in modo distribuito. Questo modulo abilita la gestione dinamica dei cluster e la resilienza a eventuali errori o modifiche, come l'elezione di nuovi leader e la fusione di cluster adiacenti. In questo modo, è possibile mantenere la consistenza dei colori e la corretta organizzazione dei nodi anche in scenari complessi.

4.4.2 Modulo `event.erl`

Il modulo `event.erl` gestisce la creazione, la gestione e il confronto degli eventi in un sistema distribuito. Gli eventi rappresentano azioni o cambiamenti di stato e sono utilizzati per mantenere la consistenza tra i componenti distribuiti. Ogni evento include informazioni come il timestamp, un ID unico, il tipo di evento, il colore associato e il PID del leader che ha generato l'evento.
Le principali responsabilità del modulo includono:

- **Creazione degli Eventi**: Il modulo fornisce funzioni per creare eventi con un timestamp attuale o con un timestamp specificato, utile per scenari di testing o ricostruzione di eventi storici.
- **Confronto degli Eventi**: Implementa una funzione per confrontare due eventi, consentendo al sistema di stabilire l'ordine temporale degli eventi in base al timestamp e, in caso di parità, all'ID unico di ciascun evento.

Le principali funzioni del modulo sono le seguenti:

- `new/3`: Crea un nuovo evento con il timestamp attuale. Gli attributi dell'evento includono il tipo, il colore e il leader che ha originato l'evento. Questa funzione è utilizzata per la generazione di eventi in tempo reale.
- `new_with_timestamp/4`: Consente la creazione di un evento con un timestamp specificato, utile in contesti di testing o per ricostruire eventi passati. L'evento conserva le stesse proprietà di `new/3`, ma accetta un timestamp definito dall'utente.
- `greater/2`: Confronta due eventi per stabilire il loro ordine temporale. Se il primo evento ha un timestamp più recente del secondo, viene considerato "maggiore". In caso di parità di timestamp, l'ID unico dell'evento è usato per determinare l'ordine. Questa funzione è essenziale per gestire correttamente la sequenza degli eventi nel sistema distribuito.

Il modulo `event.erl` è fondamentale per garantire una gestione ordinata e coerente degli eventi all'interno del sistema distribuito, assicurando che le azioni vengano processate nel corretto ordine temporale, anche in situazioni di conflitto. Utilizzando i timestamp e gli ID unici, il modulo permette al sistema di risolvere i conflitti e mantenere una visione sincronizzata dello stato distribuito.

4.4.3 Modulo `operation.erl`

Il modulo `operation.erl` fornisce una serie di funzioni per la gestione di operazioni distribuite in un sistema di cluster, con particolare attenzione alla gestione dei colori dei cluster, alla fusione di cluster adiacenti e alle operazioni di recupero in caso di guasto di un leader. Questo modulo garantisce che i nodi nei cluster mantengano una colorazione coerente e che ogni cluster abbia un leader attivo.
Le principali funzionalità del modulo sono:

- **Gestione dei Cambi di Colore**: Funzioni come `change_color/2` e `do_change_color/2` permettono di aggiornare il colore di un leader e di propagare questa informazione ai cluster adiacenti, mantenendo la consistenza dei colori. Questo è utile per evitare conflitti tra i cluster.
- **Utilità per la Fusione dei Cluster**: Funzioni come `merge_adjacent_clusters/4` e `wait_for_merge_response/4` gestiscono la fusione di cluster adiacenti quando questi condividono lo stesso colore. Queste funzioni aggiornano correttamente la lista dei nodi e dei cluster adiacenti.
- **Operazioni di Recupero**: La funzione `promote_to_leader/5` permette di promuovere un nodo a leader, utile per il recupero da guasti dei leader. In questo modo, ogni cluster ha un leader attivo, garantendo la continuità operativa e la comunicazione con i cluster adiacenti.

Le principali funzioni del modulo sono le seguenti:

- `change_color/2`: Inizia il cambiamento di colore di un leader e propaga questo aggiornamento ai cluster adiacenti, notificando anche il server. La funzione si assicura che i cambiamenti siano visibili in tutta la rete.
- `merge_adjacent_clusters/4`: Gestisce il processo di fusione tra un leader e cluster adiacenti con lo stesso colore. Assicura che i nodi e le liste di adiacenza siano aggiornati durante le operazioni di fusione.
- `remove_cluster_from_adjacent/2`: Rimuove un cluster specifico dalla lista dei cluster adiacenti, utile in caso di guasto del leader.
- `promote_to_leader/5`: Promuove un nodo a leader, inizializzando le sue informazioni di cluster e di adiacenza. È una funzione di recupero fondamentale per garantire che i cluster rimangano operativi anche in caso di guasti.
- `send_periodic_updates/1` e `broadcast_leader_update/1`: Invia periodicamente aggiornamenti sul colore e sul leader corrente a tutti i nodi del cluster e ai cluster adiacenti, garantendo una sincronizzazione continua delle informazioni.

Il modulo `operation.erl` è essenziale per il mantenimento dell'integrità e della coerenza di un sistema distribuito basato su cluster, gestendo in modo dinamico i colori dei cluster, la fusione tra cluster adiacenti e il recupero da guasti dei leader. Grazie alle sue funzioni, il sistema può adattarsi ai cambiamenti e mantenere una rete di comunicazione affidabile tra i cluster.

4.4.4 Modulo `server.erl`

Il modulo `server.erl` implementa il processo server per la gestione e il coordinamento di un sistema distribuito di nodi e cluster. Le principali responsabilità del server includono l'inizializzazione dei nodi, il monitoraggio dei leader, la gestione della consistenza dei colori dei cluster e il salvataggio dello stato del sistema in formato JSON per scopi di persistenza.

Funzionalità Principali:

- **Setup dei Nodi**: Gestisce l'inizializzazione dei nodi tramite richieste di setup, assegnando leader e aggiornando lo stato del sistema.
- **Monitoraggio dei Leader e Failover**: Monitora i leader, promuovendo nuovi leader in caso di guasti, per garantire la continuità.
- **Consistenza del Colore dei Cluster**: Verifica che i cluster adiacenti abbiano colori unici e richiede cambi di colore se necessario.
- **Persistenza dei Dati**: Salva periodicamente lo stato del server, inclusa la configurazione di nodi e leader, in file JSON per consentire il recupero dello stato del sistema.

Funzioni Principali:

- `start_server/1`: Avvia il processo server, inizializzandolo con strutture dati vuote per tracciare i nodi e i leader.
- `server_loop/4`: È il ciclo principale che gestisce i messaggi di setup, aggiornamenti dei leader, richieste di cambio colore e aggiornamenti di adiacenza.
- `start_phase2_for_all_leaders/5`: Avvia la Fase 2 per tutti i leader, raccogliendo informazioni di adiacenza e salvando la configurazione finale.
- `collect_leader_responses/3`: Raccoglie le risposte dei leader entro un timeout, confermando la loro attività nel sistema.
- `finish_setup/4`: Conclude la fase di setup, istruendo ciascun leader a salvare la propria configurazione localmente.
- `save_leader_configuration_json/1`: Converte le configurazioni dei leader in formato JSON e le salva in un file per la persistenza.
- `notify_adjacent_clusters_to_remove_cluster/2` e `notify_adjacent_clusters_about_new_leader/3`: Aggiornano i cluster adiacenti per rimuovere riferimenti a un leader terminato o aggiornare riferimenti a un leader appena promosso.

Il modulo `server.erl` è fondamentale per gestire la configurazione iniziale del sistema, monitorare i leader per prevenire guasti e assicurare che ogni cluster mantenga un colore univoco rispetto ai cluster adiacenti. Le funzioni di salvataggio in JSON permettono di mantenere uno snapshot dello stato corrente del sistema, rendendolo pronto per il debug e il recupero.

4.4.5 Modulo `setup.erl`

Il modulo `setup.erl` implementa la fase di configurazione per un sistema distribuito di nodi. Ogni nodo partecipa a un processo di configurazione per stabilire la connettività con i vicini, propagare le informazioni di leader e colore e identificare i cluster adiacenti all'interno della rete.

Funzionalità Principali:

- **Gestione della Connessione dei Nodi:** Ogni nodo stabilisce la propria connessione con i vicini, aggiornando le informazioni sul leader e la struttura del cluster.
- **Propagazione delle Informazioni di Configurazione:** Durante la propagazione, i nodi scambiano informazioni di configurazione per stabilire la gerarchia e la struttura dei leader.
- **Identificazione dei Cluster Adiacenti:** I nodi costruiscono una lista dei cluster vicini richiedendo informazioni sul leader ai nodi connessi.
- **Raccolta dei Nodi Adiacenti:** I nodi raccolgono informazioni dai vicini per stabilire una lista completa dei nodi direttamente connessi.

Funzioni Principali:

- `setup_loop/3`: Loop principale per gestire le richieste di configurazione iniziali, propagare informazioni e aggiornare lo stato di connettività.
- `setup_loop_propagate/7`: Gestisce la propagazione della configurazione, inviando richieste di configurazione ai vicini e raccogliendo gli ack.
- `wait_for_ack_from_neighbors/7`: Attende in modo ricorsivo le risposte dai vicini, aggiornando lo stato in base ai messaggi ricevuti.
- `gather_adjacent_clusters/5`: Identifica i cluster adiacenti richiedendo le informazioni sul leader ai nodi vicini.
- `gather_adjacent_nodes/3`: Raccoglie i nodi direttamente connessi costruendo una lista completa dei vicini.

Il modulo `setup.erl` è progettato per essere utilizzato in ambienti distribuiti, dove ogni nodo opera in modo indipendente ma deve collaborare con altri per formare cluster, assegnare leader e propagare i dati di connettività. La gestione dei timeout consente al modulo di rispondere a nodi non disponibili, garantendo robustezza durante la configurazione iniziale. Questo modulo si affida a funzioni di utilità come `utils:save_data/1` per gestire compiti comuni e per migliorare l'efficienza.

4.4.6 Modulo `start_system.erl`

Il modulo `start_system.erl` gestisce l'inizializzazione e la configurazione di un sistema distribuito di nodi in una griglia NxM. Fornisce funzionalità per creare un server, generare nodi con attributi di colore, assegnare i vicini e sincronizzare la configurazione iniziale assicurando che i nodi ricevano e confermino la configurazione.

Caratteristiche Principali:

- **Inizializzazione della Griglia:** Configura i nodi in una griglia NxM.
- **Assegnazione dei Colori:** Ogni nodo riceve un colore unico, caricato da un file o selezionato da una palette predefinita.
- **Assegnazione dei Vicini:** Configura le relazioni di vicinato per ciascun nodo in base alla posizione nella griglia.
- **Sincronizzazione della Configurazione:** Attende l'acknowledgment di tutti i nodi prima di iniziare la configurazione del server.
- **Salvataggio della Configurazione Finale:** Salva i dati di configurazione dei nodi in formato JSON.

Funzioni Principali:

- `start/3`: Inizializza il server e genera una griglia NxM di nodi, assegnando colori e vicini. Attende l'acknowledgment di ogni nodo prima di iniziare il setup del server.
- `load_colors_from_file/1`: Carica i colori da un file, convertendo ciascuna riga in un atomo.
- `save_nodes_data/1`: Converte i dati dei nodi in formato JSON e li salva in un file.
- `node_to_json/11`: Converte i campi di un nodo in una stringa JSON formattata.
- `find_neighbors/4`: Identifica i nodi vicini per un nodo specifico nella griglia NxM in base alle coordinate.

Precondizioni e Postcondizioni:

- **Precondizioni:** Devono essere specificate le dimensioni della griglia (N, M). Se `FromFile` è impostato su `true`, deve essere presente un file valido con i colori.
- **Postcondizioni:** Un processo server viene creato e tutti i nodi sono inizializzati con i loro colori e vicini. I dati finali della configurazione dei nodi vengono salvati in un file JSON per uso futuro.

Il modulo `start_system.erl` è utilizzato come punto d'ingresso per configurare il sistema, incluso server, nodi e vicini, e viene invocato con le dimensioni desiderate della griglia. Questo modulo supporta sia la configurazione tramite un file di colori che tramite una palette predefinita, rendendolo flessibile per vari scenari di inizializzazione.

4.4.7 Modulo `tcp_server.erl`

Il modulo `tcp_server.erl` implementa un server TCP che riceve comandi dai client per controllare un sistema distribuito di nodi. Consente la gestione di comandi come il cambio di colore di un nodo o la terminazione di un nodo specifico, elaborando e inoltrando i messaggi al processo corrispondente.

Caratteristiche Principali:

- **Server TCP:** Avvia un server TCP e ascolta su una porta disponibile.
- **Gestione Comandi:** Riceve comandi dai client, come il cambio colore o la terminazione di nodi, e li elabora.
- **Validazione Input:** Valida e converte i parametri dei messaggi ricevuti, come PID e colori.
- **Interazione in Tempo Reale:** Consente un controllo dinamico dei nodi tramite input dei client.

Funzioni Principali:

- `start/0`: Avvia il server TCP, apre una porta per ascoltare le connessioni e gestisce i client in modo concorrente.
- `listen/1`: Ascolta le connessioni TCP in arrivo e crea un processo separato per ogni client.
- `loop/1`: Gestisce il ciclo principale di ricezione dei messaggi da un client e l'inoltro della risposta.
- `handle_message/2`: Elabora un messaggio client, distinguendo comandi come il cambio di colore o la terminazione.
- `convert_to_pid/1`: Converte una stringa in un PID Erlang, se valido.
- `convert_to_color/1`: Converte una stringa in un colore della palette predefinita.
- `parse_time/3`: Converte stringhe di ore, minuti e secondi in una tupla temporale.

Il modulo `tcp_server.erl` fornisce un'interfaccia per la gestione dinamica del sistema distribuito, consentendo modifiche in tempo reale ai nodi attraverso comandi client. È ideale per ambienti in cui è necessario interagire e monitorare un sistema distribuito di nodi con feedback immediato.

4.4.8 Modulo `utils.erl`

Il modulo `utils.erl` contiene funzioni di utilità per la gestione e l'elaborazione dei dati in un sistema distribuito di nodi e cluster. Esso include operazioni su liste, formattazione, manipolazione di dati JSON, log degli eventi, e backup dei dati, promuovendo modularità e riutilizzabilità.

Caratteristiche Principali:

- **Operazioni su Liste:** Funzioni per rimuovere duplicati, unire liste, filtrare elementi e verificare condizioni specifiche su elenchi di nodi o cluster.
- **Formattazione:** Funzioni di conversione e formattazione per rappresentare colori, PID e timestamp in formati specifici.

- **Gestione JSON:** Funzioni per convertire strutture dati in JSON e salvarle su file per persistenza.
- **Logging:** Registrazione di operazioni di sistema con timestamp per facilitare il debug e la tracciabilità.
- **Backup Dati:** Funzioni per salvare i dati di nodi e leader su file, supportando il ripristino e la revisione.

Funzioni Principali:

- `remove_duplicates/1`: Rimuove elementi duplicati da una lista.
- `unique_leader_clusters/1`: Raccoglie cluster unici in base a LeaderID.
- `check_same_color/2`: Verifica se un cluster adiacente ha lo stesso colore.
- `normalize_color/1`: Converte un colore in un atomo se necessario.
- `pid_to_string/1`: Converte un PID in una stringa compatibile con JSON.
- `log_operation/1`: Registra un'operazione con un timestamp in un file di log.
- `save_leader_data_to_file/1`: Salva i dati del leader in formato JSON.
- `save_node_data_to_file/1`: Salva i dati del nodo in formato JSON.

Il modulo `utils.erl` fornisce strumenti essenziali per la gestione e il controllo del sistema, semplificando le operazioni comuni e garantendo la consistenza dei dati nei nodi e cluster distribuiti. È fondamentale per la manutenzione e la scalabilità del sistema.

4.5 Scripts

4.5.1 Script `grid_visualizer.py`

Lo script Python `grid_visualizer.py` implementa una visualizzazione grafica in tempo reale per una griglia di nodi e cluster di un sistema distribuito. Sviluppato con Flask per il server web e Matplotlib per la generazione della griglia, lo script permette agli utenti di osservare e gestire lo stato dei nodi direttamente dal browser. Grazie all'uso di Flask-SocketIO, supporta aggiornamenti in tempo reale tramite WebSocket.

Funzionalità Principali:

- **Visualizzazione della Griglia:** La griglia mostra i nodi, ciascuno colorato in base al leader di appartenenza, e connessioni visive tra i nodi dello stesso cluster, evidenziando i leader con un'etichetta speciale "L".
- **Gestione dei Colori in Tempo Reale:** Gli utenti possono modificare i colori dei nodi direttamente dall'interfaccia web. Le richieste di cambio colore vengono inoltrate al server Erlang, che aggiorna i nodi nel sistema distribuito.

- **Aggiornamenti Automatici e Logging:** La visualizzazione della griglia si aggiorna ogni 30 secondi o al rilevamento di modifiche nei dati, notificando i client via WebSocket. Ogni modifica viene registrata in un file Jupyter Notebook (`history_log.ipynb`) e salvata come snapshot.

Struttura del Codice:

- `load_leaders_data()`: Carica i dati dei leader dai file JSON, cruciali per colorare i nodi e definire i cluster.
- `draw_matrix()`: Genera la rappresentazione grafica della griglia, evidenziando leader e connessioni tra i nodi.
- `clear_notebook() e log_change_notebook()`: Gestiscono il logging delle modifiche, svuotando il notebook al riavvio e registrando le variazioni nel tempo.
- `send_color_change_request()`: Invio di richieste di cambio colore al server Erlang tramite TCP.
- `clear_snapshots_folder()`: Pulisce la cartella snapshot all'avvio per evitare sovrapposizioni di immagini.

Utilizzo:

- Avviare lo script con il comando: `python grid_visualizer.py -port <PORT> -debug <DEBUG_MODE>`.
- I parametri configurabili includono:
  - `port`: la porta TCP per comunicare con Erlang.
  - `debug`: attiva/disattiva la modalità di debug.
  - `leaders_file`, `nodes_file`, `img_path`, `notebook_path`, `snapshots_folder`: percorsi per i file JSON dei leader e dei nodi, immagine della matrice, file notebook di log, e la cartella di snapshot.

4.5.2 Script `generate_changes_rand.py`

Il file `generate_changes_rand.py` è un utile script Python progettato per gestire un sistema distribuito di nodi, permettendo di modificare colori e gestire processi in modo programmato e casuale. Lo script interagisce con un server Erlang tramite connessioni TCP, inviando comandi per modificare i colori dei nodi o terminare processi, facilitando così la gestione e il testing di configurazioni di cluster distribuiti.

Funzionalità Principali:

- **Reset del Database**: Cancella tutte le sottodirectory presenti nella cartella principale `DB` per azzerare i dati dei nodi.
- **Generazione File di Colori**: Crea un file di testo (`colors.txt`) contenente assegnazioni di colori per una griglia di nodi specificata.

- **Richieste di Cambio Colore**: Invia comandi per cambiare i colori dei nodi al server Erlang tramite socket TCP.
- **Terminazione Processi**: Identifica e termina processi in esecuzione su una specifica porta per facilitare la gestione e il setup del server.
- **Operazioni Multiple**: Esegue operazioni multiple di cambio colore e termina processi in modo casuale in base a parametri di input.

Struttura del Codice:

- `elimina_DB()`: Cancella tutte le directory nella cartella `DB` per un reset completo.
- `generate_colors_file()`: Genera un file `colors.txt` con i colori per ogni nodo della griglia, in base alle dimensioni specificate.
- `send_color_change_request()`: Invia una richiesta al server Erlang per cambiare il colore di un nodo specifico.
- `perform_multiple_operations()`: Esegue un numero definito di operazioni (cambi di colore) casuali sui nodi.
-

Utilizzo:

- Eseguire lo script con i seguenti argomenti opzionali:
  - `-rows`: Numero di righe della matrice di nodi (N).
  - `-columns`: Numero di colonne della matrice di nodi (M).
  - `-operations`: Numero di operazioni di cambio colore da effettuare.
- Esempio di utilizzo:

```
python generate_changes_rand.py --rows 5 --columns 5
                                --operations 10
```

4.5.3 Script `script.py`

Il file `script.py` è un script Python utilizzato per gestire e testare una griglia di nodi in un sistema distribuito. Lo script è particolarmente utile per verificare la reattività e la coerenza del sistema durante operazioni di cambio colore e condizioni di merge. Comunica con un server Erlang tramite TCP, consentendo di inviare richieste di cambio colore ai nodi e di eseguire scenari di test specifici per analizzare la stabilità del sistema in caso di aggiornamenti simultanei o condizioni temporali complesse.

Funzionalità Principali:

- **Reset del Database**: La funzione `elimina_DB` rimuove tutte le sottodirectory nella cartella `DB`, azzerando lo stato dei nodi per avviare un nuovo test.
- **Configurazione dei Colori**: La funzione `generate_colors_file` genera un file `colors.txt` con i colori di ciascun nodo in una griglia NxM. I colori possono essere generati casualmente o secondo una tavolozza predefinita.

Struttura del Codice:

- `elimina_DB()`: Cancella tutte le sottodirectory in `DB`, ripristinando lo stato iniziale del sistema.
- `generate_colors_file(N, M, random_colors)`: Genera un file di colori per una griglia NxM, utilizzando colori casuali o una tavolozza predefinita.
- `send_color_change_request(pid, color, timestamp, PORT)`: Invia una richiesta TCP al server Erlang per cambiare il colore di un nodo specifico.
- `get_pid_by_coordinates(data, x, y)`: Trova l'identificatore del nodo basandosi sulle coordinate specificate.
- `case1, case2, case3, too_old, change_color_during_merge, doubleMerge`: Ciascuna di queste funzioni rappresenta un caso di test diverso che simula gli scenari presentati in 3.2.7. Ciascun test e la sua struttura viene spiegato nella repository del progetto.

Utilizzo: Per utilizzare lo script, è possibile eseguirlo e seguire le istruzioni a schermo per specificare dimensioni della matrice, resettare il database e configurare la porta del server. Una volta configurato, l'utente può scegliere tra i vari casi di test da eseguire.
Esempio di utilizzo:

```
python3 script.py
```

Nota: Ogni test fornisce uno scenario unico per valutare la robustezza del sistema distribuito, testando la sensibilità ai tempi di aggiornamento, la coerenza durante le operazioni simultanee e la capacità di rispondere a merge di cluster.

4.6 File di Backup e Logging

Nel sistema distribuito progettato, vengono generati diversi file per memorizzare localmente i dati di ciascun nodo e tracciare le operazioni eseguite. Di seguito è riportata una descrizione dettagliata di ciascun file.

`data/nodes_data.json` Questo file contiene le informazioni dei nodi creati durante l'inizializzazione del sistema. Ogni nodo è rappresentato in una struttura JSON che include:

- `pid`: ID del processo Erlang che rappresenta il nodo.
- `x`: coordinata orizzontale del nodo nella griglia.
- `y`: coordinata verticale del nodo nella griglia.

Questo file è utile per tenere traccia della disposizione dei nodi e delle loro caratteristiche di base.

`data/leaders_data.json` Questo file memorizza i dati di ciascun leader, incluse le informazioni sui cluster associati. Ogni leader è rappresentato da una struttura JSON che include:

- `color`: il colore assegnato al leader, che identifica il cluster.
- `adjacent_clusters`: una lista di cluster adiacenti rappresentata da strutture `{pid, color, leader_id}`, dove `leader_id` è l'ID del leader del cluster adiacente, `pid` rappresenta un nodo vicino, e `color` è il colore associato a quel cluster.
- `nodes`: una lista degli ID di processo dei nodi appartenenti al cluster del leader.

Questo file è utilizzato per visualizzare la struttura dei cluster e le loro relazioni, rappresentando la conoscenza del server sui leader e i loro cluster.

`data/server_log.txt` Questo file registra tutte le operazioni eseguite dal server durante il funzionamento del sistema, complete di timestamp e messaggi descrittivi. Le operazioni principali comprendono:

- Ricezione e gestione delle richieste dai nodi.
- Configurazione di un nodo o di un leader.
- Avvio delle diverse fasi del sistema.

Questo log è utile per il debug e per monitorare lo stato del sistema nel tempo.

`DB/{X}_{Y}/data.json` Ogni nodo e leader ha un file in questa directory che memorizza le informazioni locali. Per un nodo normale, il file contiene le stesse informazioni di `nodes_data.json`; per un leader, include i dati di `leaders_data.json`. La struttura comprende:

- `pid`: ID del processo Erlang (per nodi normali e leader).
- `x`, `y`: le coordinate del nodo (per nodi normali e leader).
- `color`: il colore corrente del nodo (solo per leader).
- `leaderID`: l'ID del leader del cluster (per nodi normali e leader).
- `adjacent_nodes`: informazioni sui cluster adiacenti (solo per leader).

Questi file rappresentano la conoscenza locale di ciascun nodo, aggiornata continuamente durante l'esecuzione del sistema e utilizzata per mantenere aggiornato il server.

`static/matrix.png` Questo file contiene l'immagine della griglia dei nodi, generata dal modulo Python `grid_visualizer.py`. La visualizzazione rappresenta la griglia con i nodi colorati e le connessioni tra i cluster, permettendo all'utente di vedere l'organizzazione del sistema e monitorare i cambiamenti dinamici. Ogni volta che i dati di leader o nodi vengono modificati, l'immagine viene aggiornata e salvata in questo file.

Capitolo 5

Validazione

In questo capitolo vengono presentati i risultati ottenuti dall'implementazione dopo vari test eseguiti.

5.1 Ambiente di testing

I test del sistema distribuito sono stati eseguiti sulle seguenti macchine:

	Macchina 1	Macchina 2
CPU	Intel Core i7-12700K	AMD Ryzen 5 3600
GPU	NVIDIA GeForce RTX 3070	NVIDIA GeForce RTX 2060
RAM	32GB	16GB
OS	Windows 10 x64	Void Linux x86_64

E le versioni dei programmi utilizzate sono:

- Erlang/OTP 26 (Erts 14.2.5)
- rebar3 3.19.0
- Python 3.12.7
- Flask 3.0.3

Per velocizzare la compilazione del sistema distribuito in Erlang è stato scritto lo script in Bash `compile_and_run.sh`, mentre per la visualizzazione dell'interfaccia web è stato scritto lo script Python `grid_visualizer.py` basato su *Flask* per fornire un frontend con cui visualizzare la matrice di pixel e poterci interagire (approfondimenti nella sezione 4.5).

5.2 Test

5.2.1 Casistiche

Per velocizzare ed automatizzare i test delle varie casistiche possibili è stato scritto un codice ad-hoc `script.py` (spiegato in 4.5.3). I test sono stati costruiti per essere più realistici possibile e per questo è stato costruito lo script ausiliario `generate_changes_rand.py` (spiegato in 4.5.2), il quale si occupa di richiedere nuove operazioni in modo randomico, sia per quando riguarda nodi e colori, sia per la differenza nei tempi di invio e ricezione delle richieste.

Sono stati testati i seguenti requisiti:

- Scalabilità: dimensioni casuali della matrice di pixel.
- Connessione: simulazione di ritardi/perdita/duplicazione delle richieste di nuove operazioni.
- Consistenza: verifica del comportamenti in casi specifici di concorrenza menzionati in 3.2.7, per il controllo della consistenza.
- Perfomance: verifica che la complessità del sistema implementato rispetti le specifiche descritte nella sezione 3.2.2.
- Robustezza

5.2.2 Risultati

I test sono stati divisi in tre fasi:

1. la prima focalizzata sulla simulazione di un ambiente reale attraverso la generazione di eventi randomici, sia per quanto riguarda il colore scelto ed il nodo, che per i timestamp delle operazioni.
2. la seconda implementata per la verifica di serie particolari di eventi, casi di concorrenza o casi limite.
3. l'ultima per il controllo del corretto funzionamento in presenza di casi di fallimento

1) Test randomici

Si è potuto controllare il comportamento del sistema in un ambiente di utilizzo reale e "naturale". Dai risultati ottenuti si è potuto verificare che il sistema non presenza lacune, casi critici o fallimenti imprevisti. Si è potuto controllare che durante il periodo operativo non si sono presentati colli di bottiglia e tutte le operazioni sono state eseguite e terminate in tempo costante, indipendentemente dalla dimensione della matrice di pixel. Tutti i messaggi scambiati sono stati gestiti come da aspettative anche all'introduzione di ritardi nella consegna di essi o di casi di duplicazione.

2) Test inconsistenza/casi limite

Sono stati implementati test individuali e specifici per la verifica di un corretto comportamento del sistema anche introducendo inconsistenza e casi di concorrenza. Grazie alle simulazioni randomiche, è stato possibile accorgersi di alcuni casi di inconsistenza inizialmente non individuati (3.2.7), i quali sono stati risolti nell'implementazione del sistema e ricontrollati in questa stessa fase.
Anche successivamente a questi test, non sono stati rilevati comportamenti anomali o peggioramenti delle performance, pertanto si può confermare che i requisiti non funzionali (2.2) siano stati rispettati.

3) Test failure

Dato il periodo limitato per lo sviluppo del sistema, non è stato possibile condurre test specifici sui casi di fallimento (3.2.8). Va comunque osservato che il sistema è stato testato in diversi scenari e su macchine differenti e non si sono verificati mai casi di malfunzionamenti o comportamenti imprevisti del sistema distribuito implementato, pertanto possiamo confermare la generica robustezza del sistema. La mancanza di test ad-hoc per le casistiche di fallimento non implica che non siano stati implementati i meccanismi di recovery descritti nella sezione 3.2.8, ma unicamente che non è stato possibile implementare dei test che verificassero individualmente i vari casi di failure.

5.2.3 Osservazioni

Per motivi di tempistiche non è stato possibile approfondire i test sulle performance, ne ad eseguire delle ottimizzazioni alla fase di setup. Come anche osservato in 3.2.2, le performance del setup peggiorano esponenzialmente al crescere delle dimensioni della matrice e non si è potuto terminare alcun test che lavorasse su dimensioni al di sopra dei 30×30, per colpa di crash della macchina stessa. Indubbiamente con ulteriori ottimizzazioni nella fase di setup si potrebbe limitare la crescità esponenziale delle risorse utilizzate, tuttavia va anche osservato che i test sono stati condotti su singole macchine, mentre il sistema è stato sviluppato per essere eseguito in maniera distribuita.

Capitolo 6

Conclusioni