Implementazione distribuita dell'algoritmo Flood-Fill

Tam Gabriele, Merlo Filippo

12 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.

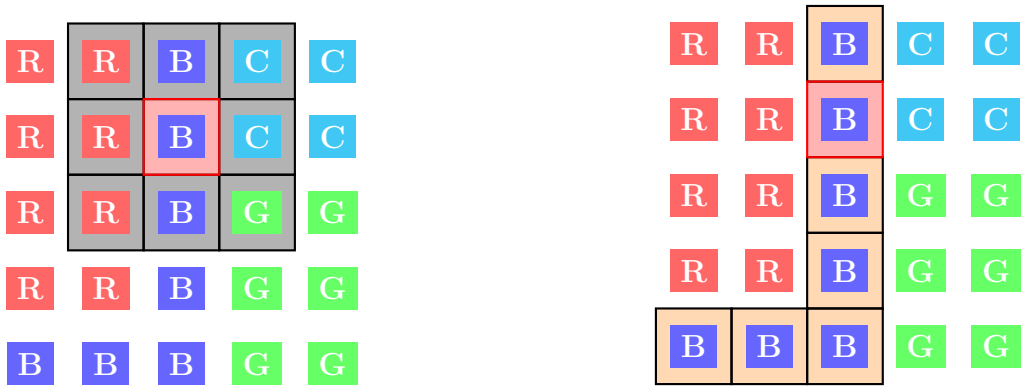
Capitolo 1

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

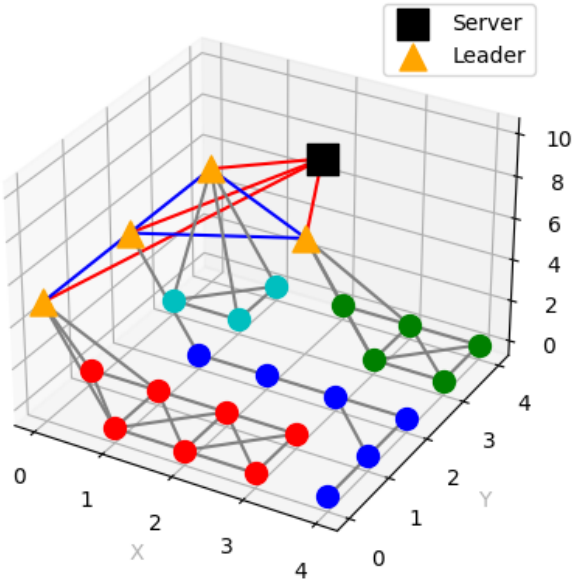


Figura 3.1: Esempio di grafo G con struttura a cluster e overlay tra i leader. Nel livello più basso sono presenti i nodi normali, disposti su una griglia 5x5 e colorati in base al cluster di appartenenza. Ogni cluster ha un leader, rappresentato da un triangolo arancione, elevato ad un livello superiore. 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`).
   - Le proprie coordinate `(x, y)` all'interno dell'immagine.
   - 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 eseguita**: Il tipo di operazione effettuata, come `merge` (fusione di due cluster), `color` (cambio di colore di un cluster), ecc.
   - **Nodi leader coinvolti**: La lista dei `leaderID` dei nodi leader che hanno partecipato all'operazione.
   - **Nuovo colore**: Il colore finale del cluster, utile nel caso di operazioni di cambio colore o merge. In caso di un'operazione `merge`, il nuovo colore sarà lo stesso dei cluster di partenza.

   {
   (timestamp, azione, [leaderID_1, leaderID_2, ...], colore),
   ...
   }

   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`): Un ID univoco che identifica il leader di un cluster.
- **Nodi nel cluster**: La lista completa dei nodi che appartengono al cluster guidato dal leader.
- **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, [leaderID_adiacente_1, leaderID_adiacente_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. Algoritmo di scambio messaggi

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

Algoritmo di inizializzazione richiesta Ogni nodo hai 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 (`colorChangeRequest`).
Il nodo scelto inizializza una richiesta di cambio colore, creando un messaggio con tag `ChangeColorRequest` assieme ai parametri necessari per la richiesta (`colore` e `timestamp`) da far recapitare al leader. Il nodo inoltra la richiesta verso il leader tramite il nodo parent.

Algorithm 1 Inizializzazione richiesta di cambio colore

```
1: procedure InitChangeColorRequest(newColor)
2:     msg ← (node.timestamp, 'colorChangeRequest', newColor);
3:     tries ← 0;
4:     do
5:         response ← await send(node.parent, msg);
6:         tries ← tries + 1
7:     while not response and tries <= 3
8: end procedure
```

Algoritmo per l'inoltro delle richieste Quando un nodo interno non leader riceve un messaggio, il suo compito è semplicemente quello di rispondere al mittente con un ACK ed inoltrare il messaggio al proprio parent. Questo meccanismo di inoltro garantisce che la richiesta raggiunga il leader, passando attraverso tutti i nodi parent nella catena di comunicazione.

Algorithm 2 Inoltro messaggio verso il leader

```
1: procedure FORWARDREQUEST(msg)
2:     send(msg.sender, ACK)
3:     tries ← 0;
4:     do
5:         response ← await send(node.parent, msg);
6:         tries ← tries + 1
7:     while not response and tries <= 3
8: end procedure
```

1.3 Algoritmo di gestione richieste del leader Il leader del cluster può ricevere vari tipi di richieste (tra cui quella di cambio colore): ciascuna richiesta viene inoltrata al server, il quale si occuperà di gestire casi particolari di concorrenza e di memorizzare l'operazione per la ridondanza.

Algorithm 3 Gestione delle richieste nodo leader

```
1: procedure LEADERRECEIVEREQUEST(msg)
2:     timestamp ← msg[0];
3:     lastTimestamp ← log[−1]['timestamp'];
4:     if timestamp > lastTimestamp then
5:         response ← await send(serverId, msg);
6:         if response then
7:             requestType ← msg[1];
8:             if requestType == 'changeColorRequest' then
9:                 node.color = msg[2];
10:                checkMerge();
11:            else
12:                ...                                  ▷ Altri tipi di richieste
13:            end if
14:        end if
15:        sync(log);
16:        sync(global_state);
17:    end if
18: end procedure
```

Il leader confronta il timestamp del messaggio con quello dell'ultima operazione eseguita salvata nel file di log. Se il timestamp del messaggio è più recente, il leader notifica il server dell'operazione ed in seguito 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 merge di cluster

Il merge tra due cluster adiacenti viene attivato quando il leader di uno dei cluster effettua un cambio di colore e scopre che il cluster adiacente ha lo stesso colore. Il leader che ha avviato il cambio colore invia la richiesta di merge al leader del cluster adiacente, indipendentemente dagli ID dei leader.

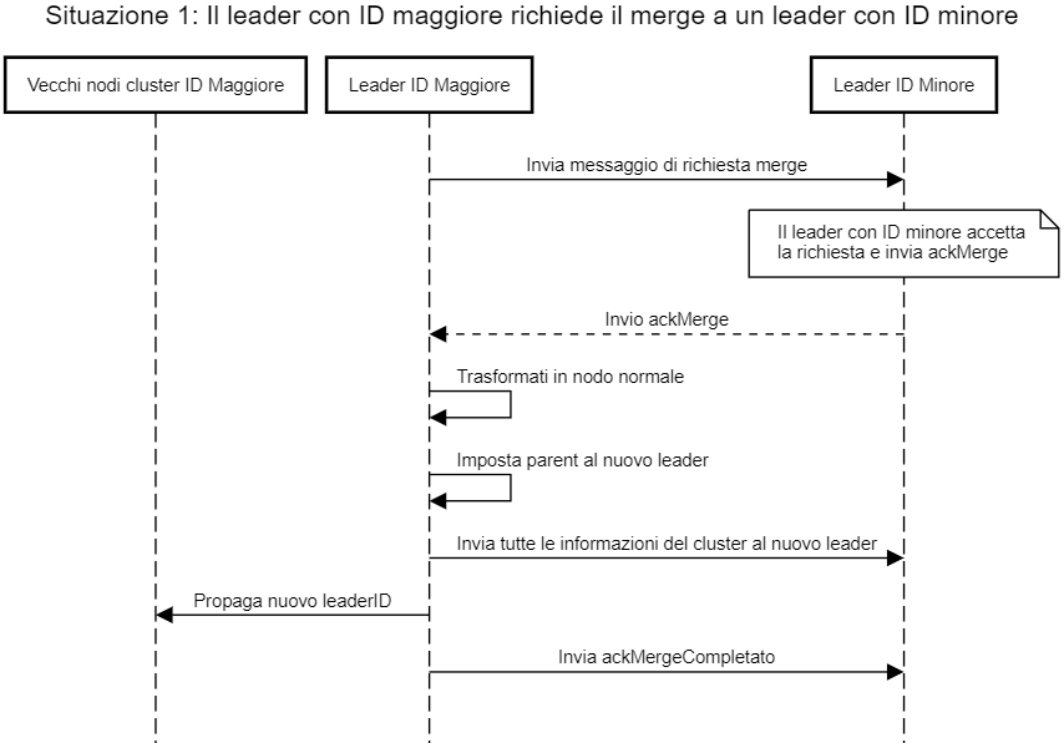
Situazione 1: richiesta di merge da leader con ID maggiore a uno con ID minore
In questo scenario, il leader con ID maggiore invia una richiesta di merge al leader del cluster con ID minore. Il leader con ID minore accetta la richiesta, e il leader con ID maggiore si trasforma in un nodo normale.

Algorithm 4 Richiesta di merge da leader con ID maggiore

```
1: procedure InvioMergeDaIDMaggiore(leaderIDMinore)
2:     response ← await send(leaderIDMinore, 'mergeRequest');
3:     if response then
4:         msg = ('leaderUpdate', leaderIDMinore);
5:         for child in node.children do
6:             send(child, msg);
7:         end for
8:         node.parent ← leaderIDMinore;
9:         node.type ← 'node';
10:        send(leaderIDMinore, node.adjList);
11:        send(leaderIDMinore, 'ackMergeCompletato');
12:    end if
13: end procedure
```

Algorithm 5 Ricezione richiesta di merge al leader con ID minore

```
1: procedure RicezioneMergeDaIDMaggiore(leaderIDMaggiore, request)
2:     if request == 'mergeRequest' then
3:         send(leaderIDMaggiore, ACK);
4:         adjListIDMaggiore ← receive();
5:         node.adjList ← join(node.adjList, adjListIDMaggiore);
6:     end if
7: end procedure
```



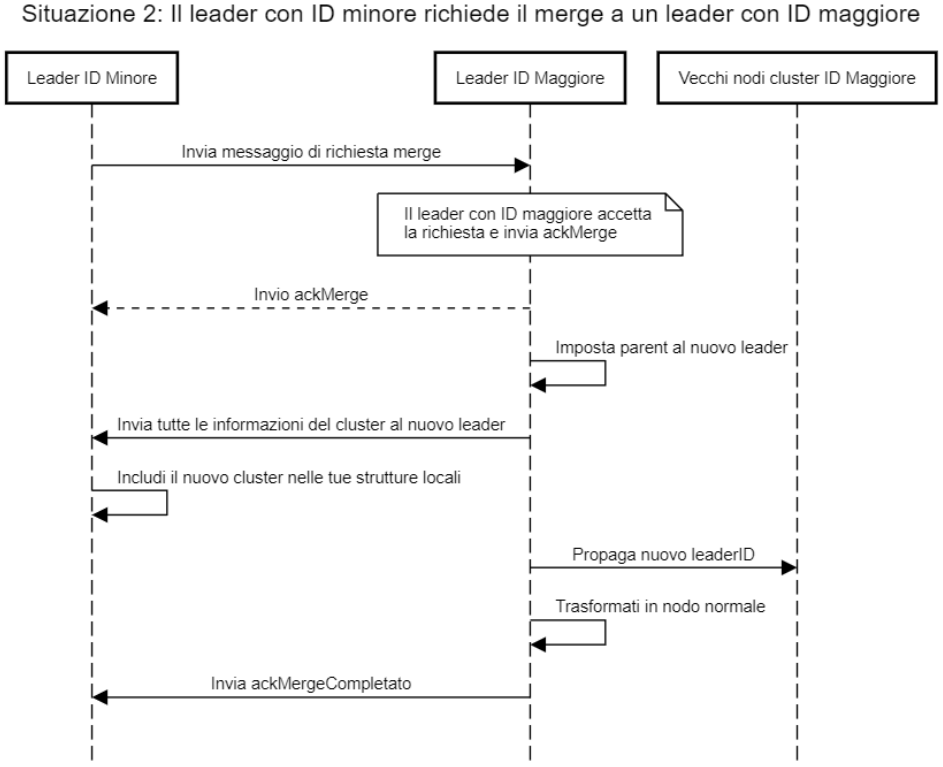
Situazione 2: richiesta di merge da leader con ID minore a uno con ID maggiore
Questo scenario è contrapposto al precedente presentato in quanto è il leader con ID minore ad inviare una richiesta di merge ad un leader adiacente con ID maggiore. Il leader con ID maggiore accetta la richiesta e si trasforma in un nodo normale.

Algorithm 6 Richiesta di merge da leader con ID minore

```
1: procedure InvioMergeDaIDMinore(leaderIDMaggiore)
2:     response ← await send(leaderIDMaggiore, 'mergeRequest');
3:     if response then
4:         send(leaderIDMaggiore, ACK);
5:         adjListIDMaggiore ← receive();
6:         node.adjList ← join(node.adjList, adjListIDMaggiore);
7:     end if
8: end procedure
```

Algorithm 7 Ricezione richiesta di merge al leader con ID maggiore

```
1: procedure RicezioneMergeDaIDMinore(leaderIDMinore, request)
2:     if request == 'mergeRequest' then
3:         msg = ('leaderUpdate', leaderIDMinore);
4:         for child in node.children do
5:             send(child, msg);
6:         end for
7:         node.parent ← leaderIDMinore;
8:         node.type ← 'node';
9:         send(leaderIDMinore, node.adjList);
10:        send(leaderIDMinore, 'ackMergeCompletato');
11:    end if
12: end procedure
```



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:

- **Messaggi di propagazione dell'ID leader**: utilizzati durante la formazione e la gestione dei cluster, questi messaggi servono a diffondere l'`id_leader` ai nodi appartenenti al cluster, assicurando che tutti conoscano il proprio leader.
- **Richieste di cambio colore (`colorChangeRequest`)**: inviate dai nodi normali verso il leader del proprio cluster quando desiderano avviare un'operazione di cambio colore. Questi messaggi contengono le informazioni sul nuovo colore e sul timestamp dell'operazione(vedi sezione 3.2.4).
- **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 merge (`mergeRequest`)**: scambiati tra i leader di cluster adiacenti durante le operazioni di unione dei cluster. Dopo la richiesta di merge, il leader che accetta l'operazione invia un `ackMerge` per confermare l'unione (vedi sezione 3.2.4).
- **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.
- **Messaggi di BFS_Request**: utilizzato durante la fase di riconfigurazione in caso di fallimento di un nodo, questo messaggio viene inviato da un nodo figlio verso i suoi vicini quando tenta di ristabilire la connettività con il leader o altri nodi del cluster. Il messaggio contiene il `leaderID` del nodo che ha avviato la ricerca e viene propagato fino a quando non si trova un nodo connesso al leader (per una spiegazione dettagliata 3.2.8).

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 (es. richiesta di cambio colore o `mergeRequest`), 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 aggiungere al sistema distribuito un server per il tempo centralizzato.
L'impiego dei timestamp 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 sul tempo di invio della richiesta. 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 il time-server invii periodicamente i timestamp aggiornati a ciascun nodo, riducendo così il sovraccarico di richieste al time-server. Ogni nodo memorizza un timestamp locale, che viene aggiornato con i dati ricevuti dal time-server. Questo approccio garantisce che i timestamp dei nodi siano sincronizzati senza la necessità di contattare continuamente il server, evitando così di sovraccaricare il sistema con troppe richieste.

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, si può utilizzare un criterio di tie-breaker, come l'ID del nodo mittente.

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 aggiornati tramite un time-server 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.
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 time-server centrale. Se il time-server dovesse fallire o diventare non raggiungibile, gli orologi dei nodi non verrebbero più sincronizzati con esso, creando il rischio di presentare divergenze tra gli orologi dei nodi, i quali causerebbero possibili inconsistenze nell'ordinamento delle operazioni. Questo problema potrebbe compromettere la correttezza del sistema e richiedere meccanismi di tolleranza ai guasti o la presenza di server di tempo ridondanti per garantire la continuità del servizio.

3.2.7 Gestione della consistenza

Per garantire la consistenza del sistema, vengono gestiti tre casi principali:

Caso 1: Richiesta di cambio colore con timestamp anteriore che richiede un merge
Se dopo un'operazione di cambio colore con timestamp T_2, arriva una richiesta con timestamp $T_1 < T_2$ che avrebbe generato un merge con un cluster adiacente all'epoca T_1 (figura 3.4):

- Si eseguono operazioni di ripristino per riportare lo stato al tempo T_1.
- Si esegue il merge tra i due cluster.
- Si applica il nuovo colore risultante dall'operazione più recente.

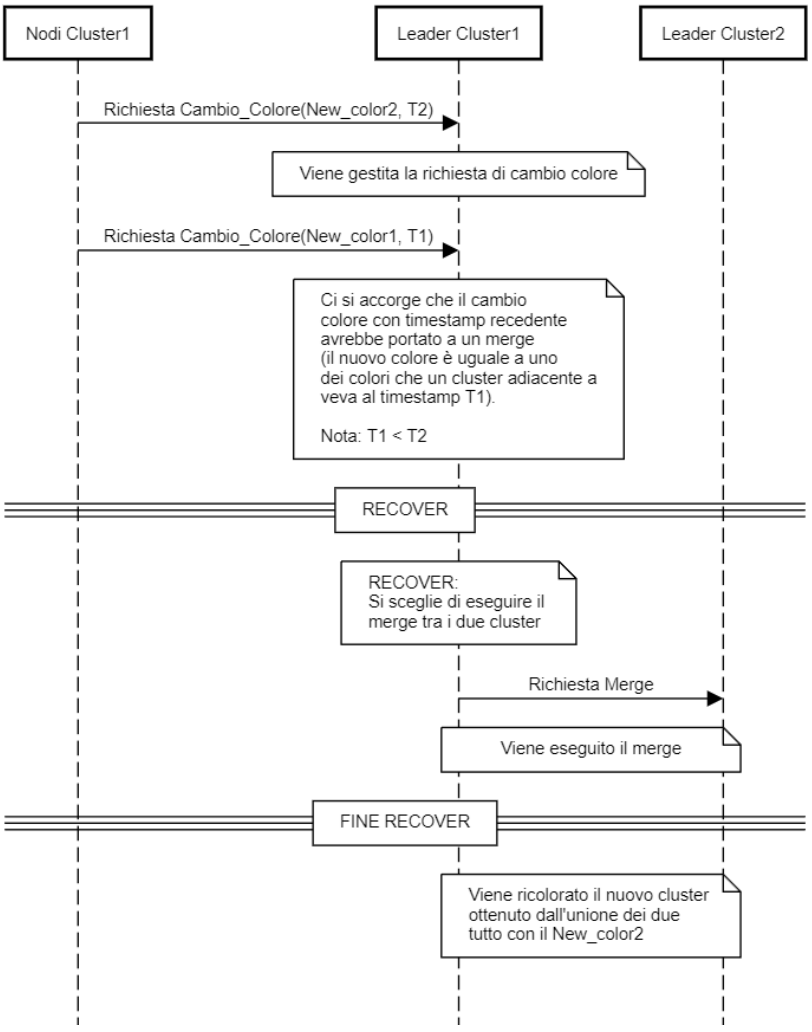


Figura 3.4: Caso 1: Richiesta di cambio colore con timestamp anteriore che richiede un merge

Caso 2: Richiesta di cambio colore con timestamp anteriore che non richiede un merge Se la richiesta con timestamp $T_1 < T_2$ non avrebbe generato un merge all'epoca T_1, ma l'operazione successiva con T_2 ha già effettuato un merge (figura 3.5):

- Si ignora l'azione con timestamp T_1 poiché lo stato attuale non corrisponde più alle condizioni dell'epoca T_1.

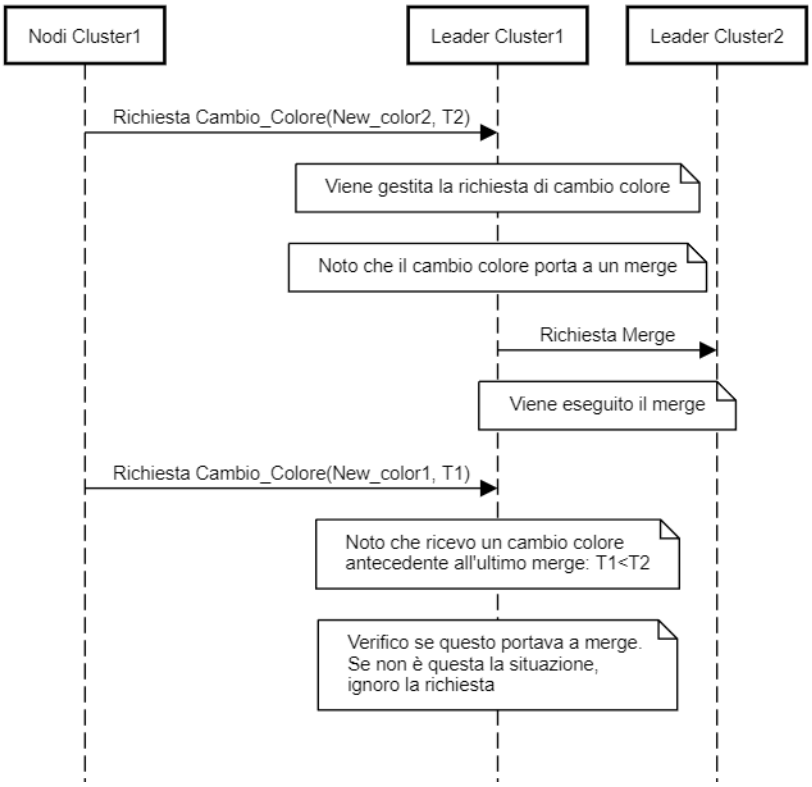


Figura 3.5: Caso 2: Richiesta di cambio colore con timestamp anteriore che non richiede un merge

Caso 3: Richiesta di merge ricevuta mentre è in corso un cambio colore con timestamp anteriore Quando un leader riceve una richiesta di merge ma è in attesa di una richiesta di cambio colore con timestamp anteriore (figura 3.6):

- Il leader attende per un periodo di tempo sufficiente per ricevere eventuali richieste in ritardo.
- Se dopo il timeout non arrivano nuove richieste, procede con il merge.
- Questo approccio previene inconsistenze dovute a operazioni fuori ordine.

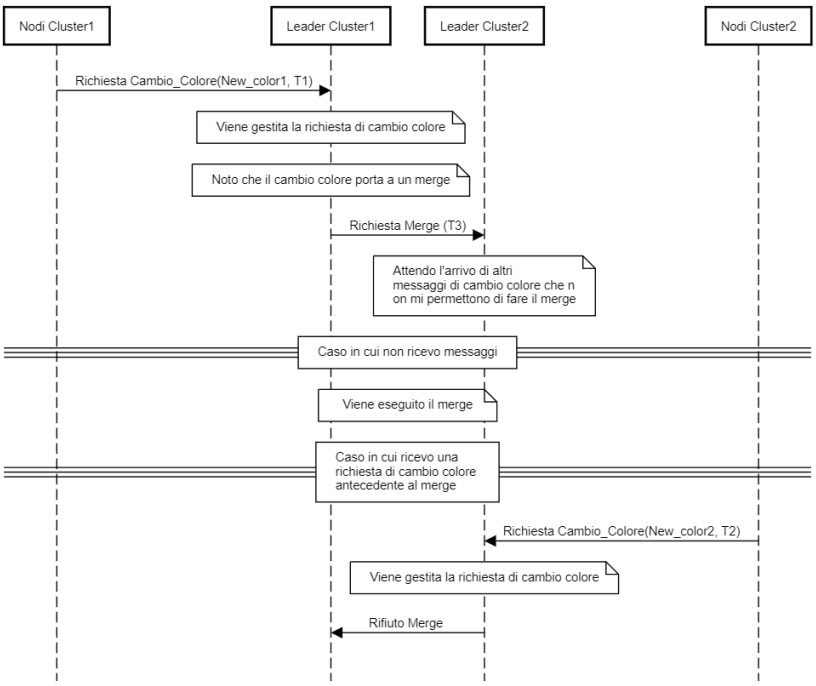


Figura 3.6: Caso 3: Richiesta di merge ricevuta mentre è in corso un cambio colore con timestamp anteriore

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.

Nel sistema descritto, si utilizza la **linearizzazione delle coordinate** per ottenere un ID univoco per ciascun nodo. Questo approccio garantisce una rappresentazione compatta ed efficiente dei nodi.

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.

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

Di seguito sono riportati i comandi da utilizzare in una sessione Erlang per compilare i moduli e avviare il sistema distribuito, seguiti dai comandi per lanciare il visualizzatore Python.

Comandi di Avvio in Erlang

```
erl
c(node).
c(event).
c(utils).
c(operation).
c(setup).
c(server).
c(tcp_server).
c(start_system).
start_system:start(10,10,false).
```

Questi comandi eseguono le seguenti operazioni:

- `c(module).` compila il modulo specificato in Erlang.
- `start_system:start(10,10,false).` avvia il sistema con una griglia di nodi di dimensioni 10x10. Il terzo parametro indica se usare un file con una lista di colori o meno. Nel caso `false`, i colori vengono scelti randomicamente da una palette.

Per avviare la parte di visualizzazione in Python, che consente d'interagire con il sistema tramite un'interfaccia grafica, è necessario eseguire il seguente comando in un'altra sessione terminale:

Avvio dell'interfaccia grafica in Python

```
python3 grid_visualizer.py
```

Questo comando esegue il file `grid_visualizer.py`, che lancia un server Flask per la visualizzazione della griglia dei nodi e la gestione dell'interfaccia utente. Il server Python si connette al sistema Erlang tramite un'interfaccia TCP, permettendo la comunicazione tra i due ambienti e l'aggiornamento dinamico della visualizzazione. È inoltre possibile fornire come opzione `--debug` per avere dettagli maggiori riguardo i nodi nell'interfaccia.

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` gestisce la creazione e gestione di ciascun nodo nel sistema. I nodi ricevono messaggi dal server e dai vicini per eseguire operazioni di propagazione, impostazione dei vicini, e cambio di colore.

Creazione di un nodo

```
new_node(X, Y) -> #node{x = X, y = Y}.
new_leader(X, Y, Color) ->
    #leader{node = new_node(X, Y), color = Color}.
```

Le funzioni `new_node` e `new_leader` inizializzano rispettivamente un nodo e un leader con specifici parametri.

4.4.2 Modulo `event.erl`

Il modulo `event.erl` contiene funzioni ausiliarie per la creazione e gestione degli eventi. In particolare la funzione `new()`, crea un nuovo evento settando il field `timestamp` al tempo corrente e il field `id` con un identificativo unico e progressivo generato da Erlang stesso.

Creazione di un nuovo evento

```
new(Type, Color, LeaderID) ->
    #event{
        timestamp = erlang:time(),
        id = erlang:unique_integer([monotonic]),
        type = Type,
        color = Color,
        from = LeaderID
    }.
```

Come si può notare, per il tempo e l'id sono state utilizzate funzioni native di Erlang, evitando così la creazione di meccanismi ad-hoc meno performanti e più complessi.

4.4.3 Modulo `utils.erl`

Il modulo `utils.erl` contiene le definizioni di funzioni ausiliarie e comuni ai vari moduli del sistema tra cui operazioni di logging, manipolazione di liste e conversione di dati.

4.4.4 Modulo `operation.erl`

Il modulo `operation.erl` contiene la definizione delle operazioni di cambio colore e di merge, comprese le sotto-operazioni che vengono eseguite dai leader per completare tali operazioni. Sono state separate dal modulo node.erl per una maggiore chiarezza e modularità del codice.

4.4.5 Modulo `setup.erl`

Il modulo `setup.erl` viene richiamato dai moduli `start_system.erl` e `node.erl` ed al suo interno sono state implementate le funzioni e i messaggi per la fase iniziale di setup del sistema, ovvero dalla scoperta dei vicini fisici, alla creazione dei cluster. Al termine dell'esecuzione del processo di setup, viene eseguito uno "snapshot" del sistema, ovvero vengono salvate le informazioni di ogni nodo in file .json separati, più viene salvata la struttura globale che viene inoltrata anche al server.

4.4.6 Modulo `server.erl`

Il server centrale, definito in `server.erl`, coordina il setup iniziale dei nodi e raccoglie le informazioni sui leader dei cluster. Ogni leader invia al server i dati dei propri nodi e dei cluster adiacenti, che vengono salvati in formato JSON per essere visualizzati esternamente.

Funzione di Logging

```
log_operation(Message) ->
    {ok, File} = file:open("server_log.txt", [append]),
    io:format(File, "~s~n", [Message]),
    file:close(File),
    io:format("LOG: ~s~n", [Message]).
```

La funzione di logging registra ogni operazione eseguita dal server su un file di log per il monitoraggio.

4.4.7 Modulo `tcp_server.erl`

Il modulo `tcp_server.erl` gestisce la comunicazione TCP, permettendo l'interazione con il modulo di visualizzazione Python per il cambio di colore dei nodi. Quando riceve un messaggio con il formato "pid, colore", tenta di aggiornare il colore del nodo e restituisce un messaggio di conferma.

Comunicazione TCP

```
loop(Socket) ->
    case gen_tcp:recv(Socket, 0) of
        {ok, BinData} ->
            [PidStr, Color] = string:split(binary_to_list(BinData),
                ",", all),
            Pid = list_to_pid(PidStr),
            gen_tcp:send(Socket, "ok");
        {error, _} -> gen_tcp:close(Socket)
```

4.4.8 Modulo `start_system.erl`

Il seguente è il modulo principale del sistema, che si occupa di spawnare i processi necessari nel corretto ordine ed avviare tutte le componenti del sistema.

4.5 Scripts

4.5.1 Script `grid_visualizer.py`

Lo script Python `grid_visualizer.py` implementa una visualizzazione grafica della griglia di nodi e cluster. Utilizza Flask e Matplotlib per generare un'interfaccia web, e supporta aggiornamenti in tempo reale grazie a WebSocket tramite Flask-SocketIO.

Visualizzazione della Matrice

```
def draw_matrix():
    leaders_data = load_leaders_data()
    nodes_data = load_nodes_data()

    fig = Figure(figsize=(8, 8))
    ax = fig.add_subplot(111)

    for node in nodes_data:
        x, y, pid = node["x"], node["y"], node["pid"]
        color = get_node_color(pid, leaders_data)
        rgb = color_to_rgb(color)
        ax.add_patch(Rectangle((y - 1, max_x - x), 1, 1,
            color=rgb, ec="black"))
```

4.6 File di backup e logging

Nel sistema distribuito progettato, vengono generati e utilizzati diversi file per mantenere memorizzati localmente i dati del sistema, nello specifico sia i dati di ogni nodo (JSON) che lo storico delle operazioni (log). Di seguito viene fornita una descrizione dettagliata di ciascun file:

- `nodes_data.json`
  Questo file contiene le informazioni sui nodi creati durante l'inizializzazione del sistema. Ogni nodo è rappresentato da 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 proprietà di base.

- `leaders_data.json`
  Questo file memorizza i dati di ciascun leader, comprese le informazioni sui cluster a cui sono 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 tuple {`pid`, `color`}, dove `pid` è l'ID del leader del cluster adiacente e `color` è il colore associato a quel cluster.
  - `nodes`: una lista d'ID di processo dei nodi che appartengono al cluster del leader.

  Questo file viene utilizzato durante la fase di visualizzazione per evidenziare la struttura dei cluster e le loro relazioni.

- `server_log.txt`
  Questo file contiene un log di tutte le operazioni effettuate dal server durante l'esecuzione del sistema. Ogni operazione è registrata con un timestamp, insieme a un messaggio descrittivo. Le operazioni tipiche includono:

  - Ricezione e gestione di richieste dai nodi.
  - Completamento della configurazione di un nodo o leader.
  - Avvio delle diverse fasi del sistema (ad esempio, Fase 1 e Fase 2).

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

- `nodes_status.txt`
  Questo file tiene traccia dello stato di ciascun nodo durante l'esecuzione. Ogni riga rappresenta un nodo e contiene informazioni come:

  - `x`, `y`: le coordinate del nodo.
  - `color`: il colore corrente del nodo.
  - `leaderID`: l'ID del leader del cluster a cui appartiene.

  Questo file è monitorato dal modulo di visualizzazione per aggiornare in tempo reale la rappresentazione grafica della griglia, riflettendo cambiamenti di stato come il colore o la posizione di un nodo.

- `leaders_data.json` (output finale della Fase 2)
  Al completamento della Fase 2, i dati finali dei leader vengono memorizzati in questo file in formato JSON. Include informazioni consolidate su tutti i leader e sui cluster adiacenti, utili per la rappresentazione della struttura del sistema una volta che l'inizializzazione è completa.

- `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 di monitorare cambiamenti dinamici. Ogni volta che i dati dei leader o dei nodi vengono modificati, l'immagine viene aggiornata e salvata in questo file.

Questi file forniscono le informazioni essenziali per il monitoraggio e la gestione del sistema distribuito, mantenendo un log dettagliato delle operazioni e fornendo dati strutturati per la visualizzazione e l'analisi.