Progetto ADCC 2023/24 Distributed Spreadsheet

Adlai Santopadre

Sommario
Il lavoro illustra i principi di progettazione adottati e le scelte del software di sincronizzazione e coerenza dei dati tra nodi distribuiti (Application OTP e MNESIA), oltre a considerare le sfide tolleranza ai guasti. Si articola in sezioni tese a descrivere il design del progetto (come mostrato a pagina intera in figura 1 di pag.4) attraverso le sezioni:
0. Disegno del progetto
1. Setup e gestione del cluster distribuito
2. Implementazione del Distributed Spreadsheet
3. API (specifiche di progetto) e analisi delle handle call relative
4. Estensione del progetto
Come materiale di studio sono stati utilizzati i testi suggeriti dal corso ADCC tenuto dal Prof.C.A. Mezzina [1] [2], la documentazione ufficiale di erlang [3] e il testo online Learning you some erlang for great good![4]

Disegno del progetto

Questo progetto riguarda lo sviluppo di un software distribuito intitolato Distributed_spreadsheet, implementato in **Erlang**, per la gestione di una cartella di lavoro distribuita composta da più schede (tabs). Ogni scheda rappresenta una tabella strutturata in **N x M celle**, corrispondenti rispettivamente alle righe e alle colonne. Il progetto sfrutta l'architettura **OTP (Open Telecom Platform)** di Erlang, utilizzando una **Distributed Application OTP** per garantire robustezza, modularità e resilienza, anche in ambienti distribuiti.

L'adozione di una **Distributed Application OTP** (si veda nella Documentazione di Sistema di Erlang Distributed Application)è una scelta basata su un sistema affidabile e conforme alle migliori pratiche di Erlang. Questa soluzione consente di separare i processi critici dall'ambiente di esecuzione. Inoltre, in un contesto distribuito, l'applicazione può gestire automaticamente la resilienza, trasferendo il carico su altri nodi in caso di fallimento di uno di essi. In altri termini, con la scelta ho perseguito di soddisfare:

- **Resilienza**: L'applicazione è gestita direttamente dal runtime di Erlang/OTP. Ciò significa che, anche in caso di crash della shell o del processo di avvio, l'applicazione persiste ed è in grado di ripristinarsi autonomamente.
- **Architettura modulare e chiara**: Una applicazione OTP segue standard ben definiti per strutturare il codice, favorendo una suddivisione logica delle funzionalità e semplificando la gestione e la manutenzione del sistema.

0.1 Struttura dell'applicazione OTP

L'architettura dell'applicazione OTP proposta si sviluppa su tre livelli principali che sono implementati con:

- **Modulo Application Callback** (`my_app`):
  - Coordina l'avvio e l'arresto dell'applicazione.
  - Ripristina spreadsheet esistenti (quelli per cui esistono metadati già salvati)
  - Si occupa di inizializzare il supervisore principale (`app_sup`) .
- **Supervisore principale** (`app_sup`):
  - Si occupa dei supervisori secondari `spreadsheet \_supervisor`.
  - Implementa una strategie di supervisione **one_for_one**, per garantire il corretto riavvio dei processi figli in caso di errori.
- **Supervisori secondari e Gen Servers**:
  - **Supervisore specifico per i fogli di calcolo** (ad esempio `spreadsheet_sup`): Gestisce uno o più processi di lavoro associati ai fogli di calcolo.
  - **Gen Server** (`distributed_spreadsheet`): Ogni Gen Server rappresenta un foglio di calcolo specifico e si occupa di gestire le operazioni principali, come la lettura e la scrittura nelle celle, le politiche di accesso e altre funzionalità legate alla gestione dei dati.

Questa architettura modulare permette di gestire efficacemente una cartella di lavoro distribuita. Ogni scheda è un'entità autonoma e isolata, ma strettamente integrata con l'applicazione intera. Questo design non solo consente una maggiore robustezza del sistema, ma facilita anche l'estensione e la manutenzione del codice nel tempo.

0.2 Codice della configurazione

La Application si avvale di parametri di configurazione specificati come liste di tuple nel file .app

Listing 1. my_app.app

```
{application, my_app,[
  {description, "Distributed Spreadsheet
      Application"},
  {vsn, "2.0"},
  {modules, [my_app, app_sup,
      spreadsheet_supervisor,
      distributed_spreadsheet]},
  {registered, [app_sup]},
  {applications, [kernel, stdlib, mnesia ]},
        % Dipendenze standard + mnesia
  {mod, {my_app, []}},
  {distributed, [{my_app, ['
      Alice@DESKTOPQ2A2FL7', '
      Bob@DESKTOPQ2A2FL7', '
      Charlie@DESKTOPQ2A2FL7']}]},
  {env, [{csv_directory, "/exported_csv"}]}
```

0.3 Uso del Database Mnesia per la persistenza dei dati

Per un sistema distribuito con resistenza ai fallimenti, ETS da solo non era sufficiente perchè ETS:

- Non è distribuito nativamente su più nodi. Le tabelle ETS risiedono solo nel nodo in cui sono state create.
- Non sono persistenti a meno che non vengano combinate con DETS, ma anche così, la persistenza è limitata a un singolo nodo.

Per soddisfare i requisiti di resistenza ai fallimenti e accesso distribuito, l'approccio definitivo è stato utilizzare Mnesia, che è il sistema di database distribuito nativo di Erlang, che incorpora sia ETS sia DETS.

Mnesia è progettato per gestire dati distribuiti, fault - tolerance e persistenza. Garantisce all'implementazione della Application OTP:

- **Distribuzione nativa**:

  Mnesia permette di replicare i dati su più nodi, garantendo che i dati rimangano disponibili anche se un nodo fallisce ed 'e possibile configurare tabelle replicate su diversi nodi, rendendo i dati accessibili anche se il nodo primario diventa inaccessibile.

- **Resistenza ai fallimenti**: In caso di crash di un nodo, Mnesia è in grado di ripristinare i dati da un altro nodo che detiene una copia della tabella. Si può configurare Mnesia per usare repliche su disco, il che significa che i dati non vanno persi anche se tutti i nodi in memoria falliscono.

- **Controllo degli accessi**: Dato che è una specifica del progetto si può implementare una politica di controllo degli accessi centralizzata tramite un nodo che funge da autorità, e anche queste politiche sono replicate sugli altri nodi per evitare un singolo punto di fallimento.

Listing 2. Codice per la creazione delle tabelle Mnesia utilizzate

```
%%% Crea le tabelle e avvia Mnesia sui nodi
    del cluster
    create_tables(Nodes) ->
        lists:foreach(fun(Node) ->
            rpc:call(Node, mnesia, start,
                []),
        end, Nodes),

    %% Crea la tabella per i dati del foglio
        di calcolo con replica
    mnesia:create_table(spreadsheet_data, [
        {attributes, record_info(fields,
            spreadsheet_data)},
        {type, bag},
        {disc_copies, Nodes},
        {index, [tab, row, col]} % Indici
            per ottimizzare le query
        ]),
    %% Crea la tabella per le politiche di
        accesso con replica
    mnesia:create_table(access_policies, [
        {attributes, record_info(fields,
            access_policies)},
        {type, bag},
        {disc_copies, Nodes}
    ]),

    %% Tabella metadati degli spreadsheet
    mnesia:create_table(spreadsheet_info, [
    {attributes, record_info(fields,
        spreadsheet_info)},
    {disc_copies, Nodes}]).
```

1. Setup e gestione del cluster distribuito

In questa sezione vengono illustrati il setup e la gestione del cluster distribuito su cui è in esecuzione l'applicazione.

Nel SO Windows 10, il setup iniziale del cluster avviene con un file `setup.bat` dedicato per creare il cluster dei nodi e predisporre una cartella riservata ai dati delle tabelle di `Mnesia`, evitando problemi di sovrapposizione.

Un modulo Erlang denominato `cluster_setup` include le funzioni necessarie per distribuire il codice compilato sui nodi. Questo modulo gestisce anche la prima inizializzazione di `Mnesia`, occupandosi della creazione dello schema, dell'avvio del database e della creazione delle tabelle replicate sui nodi. Queste tabelle sono salvate su memoria persistente per garantire la durabilità dei dati.

Con questi prerequisiti, viene avviata `my_app` che si sincronizza nel cluster ricorrendo ai file di configurazione predisposti per ciascun nodo. Il parametro `distributed` conferisce resilienza alla Application `my_app` la capacità di riavviarsi sul primo nodo disponibile in caso di fallimento del nodo in cui era attiva.

Per la gestione iniziale del cluster e il monitoraggio della rete, viene avviato un nodo di servizio denominato `monitor\_service@myhost`, che inizializza il registro dei nomi globali utilizzando un atomo generato con `list_to_atom("node"++ atom_to_list(NodeName))`. Questo atomo è associato al `Pid` di un processo locale presente su ciascun nodo del cluster, denominato `node_monitor`. Il `node_monitor` è un `gen_server OTP` implementato nel modulo omonimo, che viene caricato su tutti i nodi.

Questo processo, separato dall'applicazione principale `my_app`, consente di monitorare il cluster rilevando e gestendo i messaggi `nodedown` e `nodeup` inviati al `gen_server` in caso di arresto di uno dei nodi. È previsto il riavvio automatico del nodo interessato, durante il quale vengono registrati e riavviati sia il processo `node_monitor` sia il database `Mnesia`. Sul SO Windows 10, questa operazione avviene tramite l'esecuzione di un file `.bat` che si chiude invocando la funzione `restart_node:init()` con un modulo Erlang dedicato.

Riassumendo, i passi principali per il setup del cluster sono:

- Setup iniziale dei nodi tramite file `setup.bat`.
- Avvio del nodo di servizio e del codice per il monitoraggio del cluster (`node_monitor`).
- Inizializzazione di `Mnesia`.
- Avvio dell'applicazione my_app, configurata come distributed.
- Riavvio automatico dei nodi, quando necessario, per ristabilire l'operatività in produzione.

Nella pagina successiva in Figura 1 si mostra l'architettura.

2. Implementazione del Distributed Spreadsheet

Questa sezione descrive l'implementazione del `Distributed Spreadsheet` all'interno della Application OTP.
In particolare, affrontiamo:

- L'organizzazione di metadati e dati attraverso la definizione di tre record e delle corrispettive tabelle di `Mnesia`, create durante il setup il cui codice è già stato evidenziato.
- La parte di *Application OTP* che implementa un `gen_server` in coppia con il suo supervisore, fornendo supporto per: interagire con `my_app` tramite le API richieste, gestire più fogli di calcolo (*spreadsheet*) con caratteristiche differenti (*tabs*, righe, colonne), e ripristinare ciascuno in caso di errore tramite un supervisore **simple_one_for_one**.

2.1 Metadati e Dati: Record e Tabelle

I metadati e i dati del sistema sono organizzati tramite tre record principali:

Listing 3. File records.hrl

```
-record(spreadsheet_data, {
  name,     % Nome univoco del foglio di
      calcolo
  tab,      % indice di accesso alla tabella
  row,      %indice di accesso alla riga
  col,      %indice di accesso alla colonna
  value     %valore della cella

  }).

-record(access_policies, {
  name,  %nome univoco del foglio di calcolo
  proc,  %pid o registered name del processo
  access %accesso read o write
  }).
-record(spreadsheet_info, {
  name,     %% Nome (univoco) del foglio di
      calcolo
  rows,     %% Numero massimo di righe
  cols,     %% Numero massimo di colonne
  tabs,     %% Numero massimo di schede
  owner     %% pid del possessore

}).
```

Le tabelle di `Mnesia` sono configurate come anticipato nel modulo `cluster_setup` Ogni tabella è progettata per garantire l'efficienza con il ricorso all'indicizzazione dei campi e la coerenza dei dati distribuiti.Il legame con i record è realizzato creando le tabelle con attenzione alla generazione degli attributi in base ai campi (ad esempio):`attributes, record_info (fields,spreadsheet_data)`

2.2 Creazione dello Spreadsheet

Il processo di creazione di uno spreadsheet avviene attraverso l'interazione tra i moduli `distributed_spreadsheet` e `spreadsheet_setup`. Il codice dei passaggi principali include:

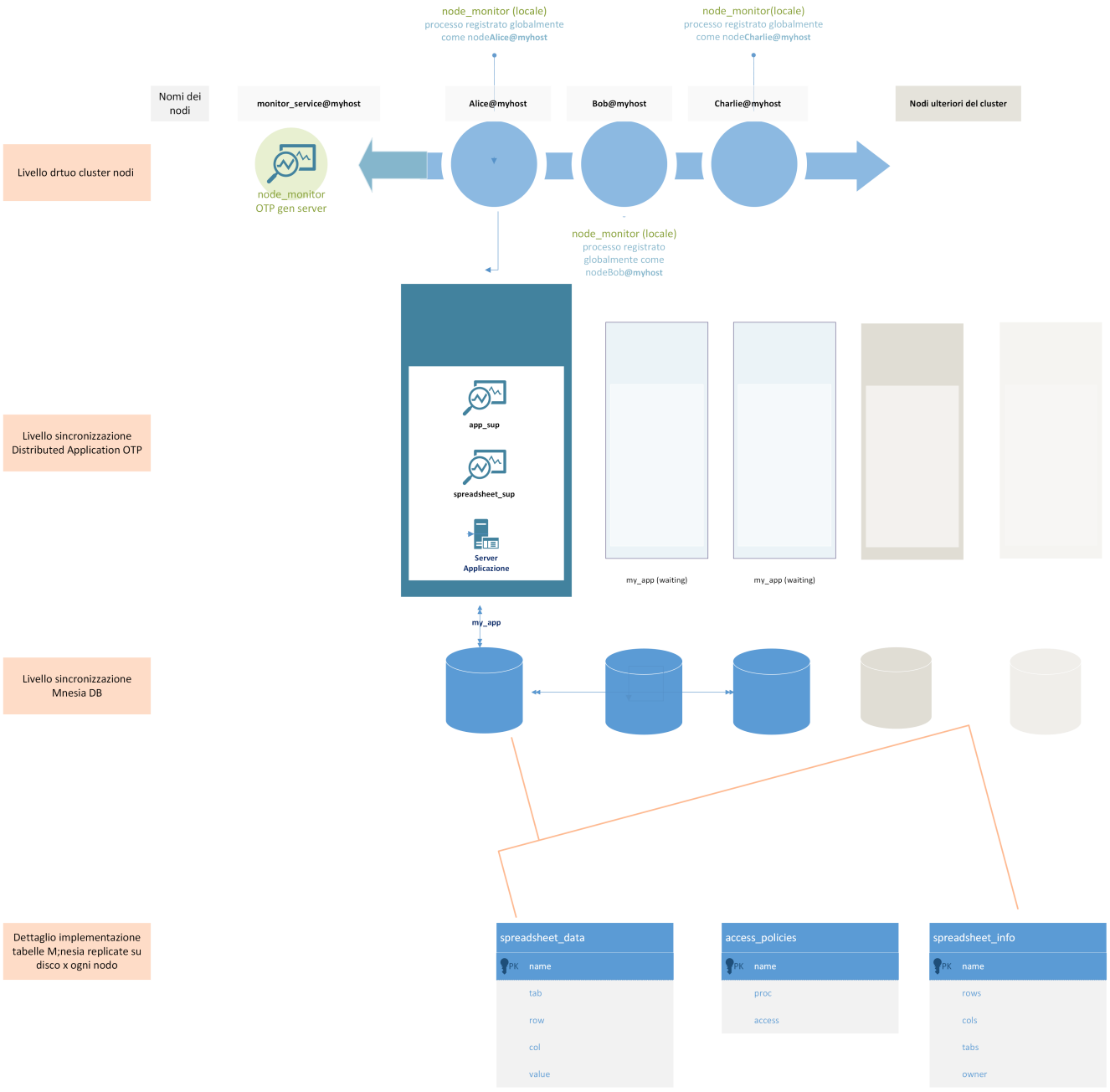

Figura 1. Architettura del foglio di calcolo

Listing 4. Creazione di un nuovo spreadsheet

```
    new(SpreadsheetName, N, M, K) when
        is_integer(N),is_integer(M),
        is_integer(K)->

    OwnerPid = global:whereis_name(
        list_to_atom("node" ++ atom_to_list(
        node()))),

    Args = {SpreadsheetName, N, M, K,
        OwnerPid},
    case spreadsheet_supervisor:
        start_spreadsheet(Args) of
        %% Invio richiesta a
            spreadsheet_supervisor per creare
             il supervisore specifico
        {ok, Pid} ->
            io:format("Spreadsheet ~p
                started successfully with PID
                 ~p~n", [SpreadsheetName, Pid
                ]),
            {ok, Pid};
        {error, Reason} ->
            io:format("Failed to start
                spreadsheet ~p: ~p~n", [
                SpreadsheetName, Reason]),
            {error, Reason}
    end.
```

Listing 5. Avvio di un child di spreadsheet_sup con passaggio dei parametri dinamico

```
%% API per avviare dinamicamente un figlio
start_spreadsheet(Args) ->
    %% Avvia un nuovo figlio per lo
        spreadsheet
    io:format("Passing params to start_child
        : ~p~n", [Args]),
    supervisor:start_child(spreadsheet_sup,
        [Args]).
```

Listing 6. Avvio del processo distributed_spreadsheet e sua registrazione globale

```
%% Avvia il gen_server /registra il nome
    globalmente
start_link(Args) ->
    io:format("Starting
        distributed_spreadsheet with args: ~p
        ~n", [Args]),
    {SpreadsheetName, _, _, _, _} = Args,
    gen_server:start_link({global,
        SpreadsheetName}, ?MODULE, Args, []).
```

3. API (specifiche di progetto) e analisi delle handle_call relative

in questa sezione darò evidenza e commento alle parti del codice che fanno riferimento all'implementazione delle API richieste nelle specifiche del progetto:

1. new/1, new/4: Creazione di nuovi spreadsheet con passaggio di parametri variabili.
2. share/2: Condivisione di spreadsheet con policy di accesso specifiche.
3. get/4, get/5: Recupero di valori da celle specifiche di uno spreadsheet.
4. set/5, set/6: Impostazione di valori in celle specifiche di uno spreadsheet.
5. info/1: Recupero di informazioni generali su uno spreadsheet.
6. to_csv/3, to_csv/2: Esportazione dello spreadsheet in formato CSV.
7. from_csv/1, from_csv/2: Importazione di uno spreadsheet da un file CSV.

Segue un'analisi del codice delle handle_call che implementano - a parte new/1 e new/4 di cui si è trattato - la logica delle API

3.1 Funzione `handle_call` - share

La funzione `handle_call/3` con il messaggio `share` è responsabile della gestione della condivisione di uno spreadsheet tra i processi con cui sono registrati i nodi.Una politica di accesso è modificata previa verifica della autorizzazione del processo chiamante

Firma: `handle\_call(\{share, SpreadsheetName, AccessPolicies, MonitorPid\},\{FromPid, Alias \},State)-> Reply`

Logica:

1. Verifica dell'autorizzazione: Viene utilizzata una transazione `mnesia` per verificare che il chiamante sia il proprietario dello spreadsheet.
   - Se non il chiamante non è autorizzato, restituisce un errore di tipo `unauthorized`.
   - Se la transazione fallisce, restituisce l'errore specifico.
2. Recupero delle politiche esistenti: In caso di autorizzazione, il codice esegue una transazione `mnesia` per ottenere le politiche di accesso esistenti associate allo spreadsheet.
   - Estrae i campi `proc` e `access` da ciascun record delle politiche esistenti.

3. Aggiornamento delle politiche di accesso: Chiama la funzione `update_access_policies` per combinare le nuove politiche con quelle esistenti.
    - Se l'aggiornamento ha successo, restituisce un messaggio di conferma.
    - In caso contrario, restituisce l'errore relativo.
4. Gestione degli errori: In caso di transazioni fallite durante la verifica o il recupero delle politiche, registra l'errore e lo restituisce al chiamante.

3.1.1 Funzione `resolve_policies`

La funzione `resolve_policies/1` preprocessa una lista di politiche di accesso, mappando ogni processo a un PID o a un nome globale risolto.

Firma: `resolve_policies (Policies) -> ResolvedPolicies`

Logica:

1. Per ogni coppia di processo (`Proc`) e politica di accesso (`Access`), tenta di risolvere il nome globale o il PID utilizzando `resolve_to_global_or_pid/1`.
2. Utilizza una mappa per aggregare i risultati, in cui le chiavi sono i processi risolti e i valori sono le politiche di accesso.

3.1.2 Funzione `resolve_to_global_or_pid`

La funzione `resolve_to_global_or_pid/1` è utilizzata per associare un identificatore di processo (`Proc`) a un PID o a un nome globale, se disponibile.

Firma: `resolve_to_global_or_pid (Proc)-> {ok, ResolvedProc} {error, Reason}`

Logica:

1. Se `Proc` è un PID:
    - Tenta di trovare un nome globale con `find_global_name/1`.
    - Se non definito, restituisce il PID originale.
2. Se `Proc` è un atomo:
    - Usa `global:whereis_name/1` per determinare la risoluzione del nome.
    - Se il nome non è trovato, restituisce un errore.

Ruolo di `Mnesia`:

- Utilizzato la tabella spreadsheet_info per verificare la proprietà dello spreadsheet.
- Recuperate e aggiornate le politiche di accesso in modo transazionale nella tabella access_policies.

3.2 Funzione `handle_call` - About

La funzione `handle_call/3` con il messaggio `about` è responsabile della raccolta di informazioni su uno spreadsheet specifico, comprese le politiche di accesso e le caratteristiche generali.

Firma: `handle_call({about, SpreadsheetName}, From, State) -> Reply`

Logica:

1. Recupero delle informazioni dello spreadsheet: utilizza una transazione `mnesia` per cercare il record corrispondente nella tabella `spreadsheet_info`.
    - Se trovato, calcola il numero totale di celle per tab (`CellsxTab`).
    - Se non trovato, restituisce un errore di tipo `spreadsheet_not_found`.
2. Recupero delle politiche di accesso: Esegue una transazione `mnesia` per ottenere i permessi di lettura e scrittura associati allo spreadsheet.
    - Filtra i permessi per distinguere tra lettura e scrittura.
3. Creazione della mappa dei risultati: Compila una mappa `Info` contenente:
    - Nome dello spreadsheet.
    - Proprietario.
    - Numero totale di tab.
    - Numero totale di celle.
    - Permessi di lettura e scrittura.
4. Gestione degli errori: Registra e gestisce eventuali errori durante le transazioni.

3.3 Funzione `handle_call` - Get

La funzione `handle_call/3` con il messaggio `get` è utilizzata per recuperare il valore di una specifica cella di uno spreadsheet, verificando i permessi del chiamante.

Firma: `handle_call({get, SpreadsheetName, TabIndex, I, J, MonitorPid, From, State)-> Reply`

Logica:

1. Log iniziale: Registra i dettagli della richiesta, inclusi il tab, la riga e la colonna richiesti dal processo chiamante.
2. Verifica dei permessi: Utilizza la funzione `check_access/3` per assicurarsi che il chiamante abbia accesso almeno in lettura allo spreadsheet.
    - In caso di permesso negato, restituisce un errore di tipo `access_denied`.

3. Recupero del valore: Se i permessi sono validi, esegue una transazione `mnesia` per cercare il valore nella cella specificata:
    - se trovato, restituisce il valore.
    - se non trovato, restituisce `undef` che è un valore di inizializzazione per i valori dello spreadsheet.
    - Registra eventuali errori di transazione e restituisce un errore relativo.
4. Gestione degli errori: Gestisce e registra eventuali errori durante il recupero del valore o la verifica dei permessi.

Esempio di Log:
```
Get request from ... for Tab: ..., Row:..., Col:...
Returning value for Tab: ..., Row: ..., Col: ...
Transaction aborted for get request: ...
Access denied for process ...
```

3.4 Funzione `handle_call` - Set

La funzione `handle_call/3` con il messaggio `set` consente di aggiornare il valore di una cella specifica di uno spreadsheet, verificando i permessi di scrittura del chiamante.

Firma: handle_call({set, SpreadsheetName, TabIndex, I, J, MonitorPid, Value}, _From, State)} `-> Reply`

Logica:

1. Log iniziale: Registra i dettagli della richiesta, inclusi il tab, la riga, la colonna e il nuovo valore da impostare.
2. Verifica dei permessi: Utilizza `check_access/3` per assicurarsi che il chiamante abbia accesso in scrittura allo spreadsheet.
    - In caso di permesso negato, restituisce un errore di tipo `access_denied`.
3. Aggiornamento del valore: Se i permessi sono validi, esegue una transazione `mnesia` per aggiornare il valore della cella specificata.
    - Recupera eventuali record esistenti corrispondenti alla cella e li elimina.
    - Scrive un nuovo record con il valore aggiornato.
    - Registra il risultato della transazione e restituisce il valore aggiornato.
4. Gestione degli errori: Gestisce e registra eventuali errori durante l'aggiornamento del valore o la verifica dei permessi.

validazione di Value Prima di essere passato alla handle_call in esame il valore Value viene controllato e validato dalla funzione validate_value/1 dalla API `set`

3.5 Funzione `handle_call` - To CSV

La funzione `handle_call/3` con il messaggio `to_csv` consente di esportare i dati di uno spreadsheet in un file CSV, garantendo l'integrità dei dati e la corretta scrittura del file.

Firma: `handle_call({to_csv, SpreadsheetName, Filename}, From, State) -> Reply`

Logica:

1. Verifica del nome dello spreadsheet: Confronta il nome dello spreadsheet richiesto con quello presente nello stato.
    - Se il nome non corrisponde, restituisce un errore `spreadsheet_not_found`.
2. Recupero dei dati: Utilizza una transazione `mnesia` per leggere tutti i record relativi allo spreadsheet.
3. Scrittura del file CSV: Utilizza la funzione `write_csv` per scrivere i record nel file Filename specificato.
    - Se la scrittura è completata con successo, restituisce un messaggio di conferma.
    - In caso di errore, restituisce il motivo del fallimento.

Esempio di Log:

```
Exporting spreadsheet ... to file ...
Spreadsheet ... to be exported to ...
Record to be written ...
Spreadsheet ... exported successfully to ...
Failed to write CSV for spreadsheet ...
Transaction aborted during export: ...
```

Ruolo di `write_csv`:

- Scrive i dati recuperati in un file CSV specificato dal chiamante aggiungendo due righe di intestazione con i metadati.
- Gestisce eventuali errori durante la scrittura del file.

3.6 Funzione `handle_call` - From CSV

La funzione `handle_call/3` con il messaggio `from_csv` consente di importare dati da un file CSV in uno spreadsheet da implementazione, aggiornando le tabelle e garantendo la consistenza dei dati.

Firma: `handle_call({from_csv, Filename, Name}, From, State) -> Reply`

Logica:

1. Lettura del file CSV: Utilizza la funzione `read_from_csv` per leggere i record dal file specificato.

- Se la lettura ha successo,attraverso una catena di funzioni ausiliarie che analizzano il contenuto del file (ricavando i metadati dalle prime righe di intestazione) vengono estratti i record rappresentati da ogni successiva riga interpretata come record strutturato nella forma di #spreadsheet_data.
- In caso di errore, restituisce il motivo del fallimento.

2. Salvataggio dei dati: Esegue una transazione `mnesia` per aggiornare lo spreadsheet con i dati importati.
   - Elimina i record esistenti relativi allo spreadsheet dalla tabella `spreadsheet_data`.
   - Scrive i nuovi record nella tabella utilizzando `mnesia:write/1`.
3. Gestione della transazione: Verifica il risultato della transazione:
   - Se completata con successo, restituisce un messaggio di conferma.
   - Se fallisce, restituisce il motivo del fallimento.

Esempio di Log:

```
Records fetched: ...
Transaction aborted: ...
```

4. Estensione del progetto

Nella versione avanzata di `distributed spreadsheet` viene richiesto di implementare ulteriori API. La possibilità di inserire nuove righe e di cancellare righe intere può essere implementata con transazioni Mnesia di tipo atomico, avendo cura di modificare anche i metadati relativi al numero di righe N che per lo spreadsheet da modificare è disponibile nella tabella spreadsheet_info. La possibilità di estendere alle Macro i valori accettati in Value può essere gestita ampliando i controlli di validazione dei dati e passando alle celle l'espressione contenuta nella direttiva -define della Macro per poi recuperarla ad uso anche dinamico. Così si può implementare anche operazioni che utilizzano più valori adiacenti di una riga N dalla colonna Kespr alla colonna Kresult utilizzandoli come valori parametrici per costruire una funzione il cui valore viene ritornato nella Kresult con parametri contenuti nelle colonne della riga N comprese tra Kespr a Kresult.

Riferimenti bibliografici

[1] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, Dallas, TX, 2nd edition, 2013.

[2] Francesco Cesarini and Simon Thompson. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media, Sebastopol, CA, 2011.

[3] Erlang/OTP Team. Erlang documentation, 2025. Accessed: 2025-01-15.

[4] Fred Hébert. Learning you some erlang for great good!, 2011. Accessed: 2025-01-15.