Università degli studi di Urbino Carlo Bo
Corso di laurea in informatica applicata

Relazione Progetto D'Esame

ADCC: Applicazioni Distribuite e Cloud Computing

Sessione invernale - A.A. 2023/2024

Professore: Claudio Antares Mezzina

Studenti:
Giacomo Di Fabrizio
Matteo Marco Montanari

17 Gennaio 2024

Specifica Di Progetto

Il progetto prevede la creazione di un'applicazione distribuita, scritta in linguaggio Erlang, che per permetta di gestire dei fogli di calcolo (similmente al software Microsoft Excel).
Questo strumento consentirà agli utenti di generare fogli di calcolo, ciascuno dei quali potrà contenere diverse tabelle. All'interno di queste tabelle, le celle saranno accessibili per la lettura o la scrittura a chiunque abbia i relativi permessi.
Gli utenti avranno la possibilità di condividere ciascun foglio di calcolo con gli altri utenti in modalità di sola lettura o di scrittura (e lettura).

Modellazione del problema

Per poter realizzare l'applicazione, si è deciso di dividerla in più moduli, ognuno con uno scopo ben preciso.

Modulo *spreadsheet*

Ogni foglio di calcolo può essere rappresentato tramite una matrice di $M \times N$ celle, dove ogni cella può contenere:

- Un qualsiasi tipo di dato primitivo
- Il valore dell'atomo *undef*

Il modulo contiene le seguenti funzioni:

- new(Name) → Spreadsheet | {error, Reason}
  - Crea un nuovo foglio di nome "Name" di dimensioni $M \times N$ di K tabelle, dove:
  - I parametri N, M, K sono valori definiti di default nel modulo stesso.
  - Assegna il processo creatore come proprietario del foglio.
- new(Name, N, M, K) → Spreadsheet | {error, Reason}
  - Crea un nuovo foglio di nome "Name" di K tabelle.
  - Ogni tabella ha dimensioni $M \times N$.
  - Assegna il processo creatore come proprietario del foglio.
- share(Spreadsheet, AccessPolicies) → bool
  - Il proprietario del foglio può condidivere il foglio in lettura o scrittura con altri processi. Solo se il foglio è condiviso allora può essere acceduto da altri utenti.
  - AccessPolicies è una tupla {Proc, AP} dove:
    - Proc è un Pid o nome registrato (reg_name).
    - AP := read | write
  - Le policy di accesso ad un foglio possono cambiare in qualsiasi momento in maniera dinamica.

- get(Spreadsheet, Tab, I, J) → Value | undef
  - Legge il valore della cella (I, J) che appartiene alla tabella Tab del foglio Spreadsheet
- get(Spreadsheet, Tab, I, J, Timeout) → Value | undef | timeout
  - Come la funzione *get* definita sopra ma con un valore di timeout (in millisecondi) entro il quale l'esecuzione termina sicuramente.
- set(Spreadsheet, Tab, I, J, Val) → bool
  - Scrive il valore Val nella cella (I, J) che appartiene alla tabella Tab del foglio Spreadsheet.
- set(Spreadsheet, Tab, I, J, Val, Timeout) → bool | timeout
  - Come la funzione *set* definita sopra ma con un valore di timeout (in millisecondi) entro il quale l'esecuzione termina sicuramente.
- info(Name) → Spreadsheet_info
  - Ritorna le informazioni del foglio Name, in particolare:
    - Il numero di celle per ogni tabella all'interno del foglio Name.
    - I permessi di lettura e scrittura del foglio Name (quali processi hanno i vari permessi).

Modulo *csv_manager*

Questo modulo è specificamente progettato per facilitare le conversioni tra fogli e file CSV in entrambe le direzioni.
Il modulo contiene le seguenti funzioni:

- to_csv(Spreadsheet, Filename) → ok | {error, Reason}
  - Esporta in formato CSV il foglio Spreadsheet con il nome "Filename.csv".
- to_csv(Spreadsheet, Filename, Timeout) → ok | {error, Reason} | timeout
  - Come la funzione *to_csv* definita sopra ma con un valore di timeout (in millisecondi) entro il quale l'esecuzione termina sicuramente.
- from_csv(Filename) → Spreadsheet | {error, Reason}
  - Ricrea il foglio a partire dal file CSV corrispondente.
- from_csv(Spreadsheet, Filename, Timeout) → Spreadsheet | {error, Reason} | timeout
  - Come la funzione *from_csv* definita sopra ma con un valore di timeout (in millisecondi) entro il quale l'esecuzione termina sicuramente.

Modulo *distribution*

Questo modulo è progettato per la creazione di tabelle, l'avvio e la terminazione del DBMS Mnesia in ambiente sia locale che distribuito.
Il modulo contiene le seguenti funzioni:

- create_table() → {atomic, ok} | {error, Reason}
  - Crea le tabelle del Database nel nodo locale (in memoria e su disco).

- create_table_distrib() → {atomic, ok} | {error, Reason}
  - Crea le tabelle del Database (distribuito) nel nodo locale e nei nodi remoti (in memoria e su disco). I nodi remoti sono tutti i nodi conosciuti da colui che invoca questa funzione.
- start() → ok | {error, Reason}
  - Avvia il DBMS Mnesia nel nodo locale.
- start_distrib() → ok | {error, Reason}
  - Avvia il DBMS Mnesia nel nodo locale e nei nodi remoti. I nodi remoti sono tutti i nodi conosciuti da colui che invoca questa funzione.
- stop() → ok | {error, Reason}
  - Termina il DBMS Mnesia nel nodo locale.
- stop_distrib() → ok | {error, Reason}
  - Termina il DBMS Mnesia nel nodo locale e nei nodi remoti. I nodi remoti sono tutti i nodi conosciuti da colui che invoca questa funzione.

Modulo *debug*

Questo modulo è stato progettato per il debugging e il testing dell'applicazione da noi realizzata. Per semplicità, le funzioni di testing sono state pensate per essere eseguite in locale ma sono facilmente adattabili ad un contesto distribuito.
Il modulo contiene le seguenti funzioni:

- delete_close_tables() → ok
  - Elimina tutte le tabelle Mnesia dal nodo locale. Funzione di debug.
- spawn_reader(Spreadsheet, Pid) → PidReader
  - Crea un processo che legge un valore dal foglio Spreadsheet e lo invia al processo Pid. Restituisce il pid del processo spawnato. Funzione di testing.
- spawn_writer(Spreadsheet, Pid) → PidWriter
  - Crea un processo che scrive un valore dal foglio Spreadsheet e lo invia al processo Pid. Restituisce il pid del processo spawnato. Funzione di testing.
- test_reading(PidLettori) → TestResult,
  - Avvia un test di lettura tramite i processi spawnati dalla funzione *spawn_reader*. Restituisce i risultati delle letture. Funzione di testing.
- test_writing(PidScrittori)→ TestResult,
  - Avvia un test di scrittura tramite i processi spawnati dalla funzione *spawn_writer*. Restituisce i risultati delle scritture. Funzione di testing.
- test_sequential(N, Foglio) → {{readers, PidLettori, TestLetture}, {writers, PidScrittori, TestScritture}},
  - Avvia un test (sul foglio Foglio) costituito da una lettura e una scrittura per ognuno degli N processi spawnati. Restituisce i risultati delle letture e scritture per ogni processo. Funzione di testing.

- test_mixed(N, M, Foglio) → {{readers, PidLettori}, {writers, PidScrittori}, {results, Tests}},
  - Avvia una serie di test (sul foglio Foglio) costituita da M operazioni, fatte da una lettura seguita da una scrittura, per ognuno degli N processi spawnati. Restituisce i risultati di ogni operazione per ogni processo. Funzione di testing.
- timing_ms(TestName, Params) → Time,
  - Avvia il test TestName sui parametri Params e restituisce il tempo impiegato ad eseguire il test in millisecondi. Funzione di testing.
- test_battery(Foglio) → {{sequential_test, {params, ParamsSequential}, {time_ms, TimeSequential}}, {mixed_test, {params, ParamsMixed}, {time_ms, TimeMixed}}, {stress_mixed_test, {params, ParamsStress}, {time_ms, TimeStress}} },
  - Esegue una batteria di test (sul foglio Foglio) con parametri di default calcolando il tempo impiegato ad eseguire i vari test. La batteria è composta da: un test sequenziale (test_sequential), un test misto (test_mixed) e uno stress test. Funzione di testing.
- average_lookup_time(Foglio, N) → {microsecond, AverageLookup},
  - Esegue una serie di N letture sul foglio Foglio restituendo il tempo medio (in microsecondi) impiegato ad effettuare una singola lettura. Funzione di testing.
- average_setup_time(Foglio, N)→ {microsecond, AverageSetup},
  - Esegue una serie di N scritture sul foglio Foglio restituendo il tempo medio (in microsecondi) impiegato ad effettuare una singola scrittura. Funzione di testing.

Scelte implementative

Per la realizzazione dell'applicazione distribuita, si è deciso di utilizzare Mnesia.
Mnesia è un sistema di gestione di database (DBMS) distribuito e transazionale.
Questo database è progettato per essere altamente affidabile, scalabile e resistente alle fallimenti, elementi critici in ambienti distribuiti.
Alcune caratteristiche chiave di Mnesia includono:

- Distribuzione e Concorrenza:
  - Mnesia è progettato per funzionare su sistemi distribuiti, consentendo la distribuzione di dati su più nodi. Inoltre, gestisce in modo efficiente situazioni concorrenti, grazie al modello di concorrenza Erlang, che è basato su processi leggeri e isolati.
- Transazionalità:
  - Mnesia supporta transazioni, consentendo operazioni atomiche su un insieme di dati. Questo è cruciale per garantire la coerenza dei dati, specialmente in ambienti distribuiti.
- Persistenza:
  - I dati memorizzati in Mnesia sono persistenti, il che significa che sopravvivono anche dopo che il sistema è stato spento o riavviato.
- Schema Dinamico:
  - A differenza di molti sistemi di database relazionali, Mnesia permette uno schema dinamico. Ciò significa che è possibile modificare la struttura del database senza dover interrompere il sistema o apportare modifiche complesse allo schema.

Per poter realizzare in maniera corretta l'applicazione, si è deciso di considerare ogni foglio come una tabella mnesia.
In particolare, ogni foglio possiede la seguente struttura:

Record Name	table	riga	colonne
foglio1	1	1	[undef,undef,undef,undef]
foglio1	1	2	[undef,undef,undef,undef]
foglio1	1	3	[undef,undef,undef,undef]
foglio1	2	1	[undef,undef,undef,undef]
foglio1	2	2	[undef,undef,undef,undef]
foglio1	2	3	[undef,undef,undef,undef]
foglio1	3	1	[undef,undef,undef,undef]
foglio1	3	2	[undef,undef,undef,undef]
foglio1	3	3	[undef,undef,undef,undef]
foglio1	4	1	[undef,undef,undef,undef]
foglio1	4	2	[undef,undef,undef,undef]
foglio1	4	3	[undef,undef,undef,undef]
foglio1	5	1	[undef,undef,undef,undef]
foglio1	5	2	[undef,undef,undef,undef]
foglio1	5	3	[undef,undef,undef,undef]

Figura 1: Struttura della tabella foglio

Come è possibile notare dalla figura 1, ogni foglio è rappresentato da una tabella mnesia di tipo *bag* (in quanto consente la duplicazione delle chiavi) popolata da record di tipo:
$-record(spreadsheet, \{table, riga, colonne\})$.
In particolare:

- Nel campo *table* verrà inserito un intero che identifica il numero della tabella all'interno del foglio
- Nel campo *riga* verrà inserito un intero che identifica il numero della riga della tabella specificata dal campo *table*
- Nel campo *colonne* verrà inserita una tupla contenente tanti valori quanti sono il numero degli elementi presenti nelle colonne corrispondenti alla riga specificata dal campo *riga*

Nella figura 1, è mostrato un foglio (chiamato *foglio1*) che possiede 5 tabelle, ciascuna di 3 righe e 4 colonne. Tutte le celle del foglio sono riempite col valore *undef*.

Per consentire la memorizzazione del proprietario del foglio, è stata utilizzata un'ulteriore tabella mnesia che chiamiamo *owner*.
Come è possibile notare dalla figura 2, questa è una tabella con una chiave primaria univoca (campo *foglio*) popolata da record di tipo:
$-record(owner, \{foglio, pid\})$.
In particolare:

- Il campo $foglio$ è la chiave primaria della tabella. Qui verrà inserito il nome di ciascun foglio.
- Nel campo *pid* verrà inserito il pid (che funge da identificativo dell'utente) proprietario del foglio.

Record Name	foglio	pid
owner	foglio1	<0.85.0>
owner	foglio2	<0.16917.0>
owner	foglio3	<0.16988.0>
owner	foglio4	<0.17070.0>
owner	foglio5	<0.17079.0>

Figura 2: Struttura della tabella owner

La struttura della tabella *owner* garantisce che, tramite un controllo su quest'ultima, solo il proprietario del foglio potrà condividere tale foglio con altri utenti a sua scelta.

Per consentire la memorizzazione dei permessi che un utente possiede nella visualizzazione o nella scrittura delle celle di un foglio, è stata realizzata la tabella mnesia chiamata *policy*, la cui struttura è rappresentata in figura 3.

Record Name	pid	foglio	politica
policy	<0.85.0>	foglio1	write
policy	<0.127.0>	foglio6	read
policy	<0.129.0>	foglio6	read
policy	<0.130.0>	foglio6	read
policy	<0.131.0>	foglio6	write
policy	<0.16917.0>	foglio2	write
policy	<0.16988.0>	foglio3	write
policy	<0.17070.0>	foglio4	write
policy	<0.17079.0>	foglio5	write
policy	<0.17079.0>	foglio6	write

Figura 3: Struttura della tabella policy

Come è possibile notare dalla figura 3, questa è una tabella di tipo *bag* popolata da record di tipo: $-record(policy, \{pid, foglio, politica\})$.
In particolare:

- Nel campo *pid* verrà inserito il pid dell'utente a cui è stato condiviso un foglio
- Nel campo $foglio$ verrà inserito il foglio condiviso tra il suo proprietario e l'utente specificato dal campo *pid* corrispondente
- Nel campo *politica* verrà inserita la politica di condivisione del foglio per quanto riguarda l'utente specificato dal campo *pid* corrispondente.

Il campo *politica* potrà contenere il valore *write* o il valore *read* in mutua esclusione.
Se nel campo è presente il valore *read* l'utente, identificato con il *pid* potrà leggere qualsiasi cella del $foglio$.
Se nel campo è presente il valore *write* l'utente, identificato con il *pid* potrà sia scrivere che leggere qualsiasi cella del $foglio$.

La struttura della tabella *policy* permette il fatto che, tramite controllo su quest'ultima, solo gli utenti autorizzati abbiano il permesso di operare sul foglio condiviso.
In aggiunta, questa tabella agevola la visualizzazione di tutti gli utenti autorizzati sia alla scrittura che alla lettura su un particolare foglio. Ovviamente, il processo che ha creato il foglio condiviso, sarà presente all'interno di questa tabella con il permesso write.

Infine, la tabella mnesia chiamata *format* è stata utilizzata per poter memorizzare il formato di ogni foglio. La struttura di questa tabella è mostrata in figura 4.

Record Name	foglio	tab_index	nrighe	ncolonne
format	foglio1	1	3	4
format	foglio1	2	3	4
format	foglio1	3	3	4
format	foglio1	4	3	4
format	foglio1	5	3	4
format	foglio2	1	3	4
format	foglio2	2	3	4
format	foglio2	3	3	4
format	foglio2	4	3	4
format	foglio2	5	3	4
format	foglio3	1	3	4
format	foglio3	2	3	4
format	foglio3	3	3	4
format	foglio3	4	3	4
format	foglio3	5	3	4
format	foglio4	1	3	4
format	foglio4	2	3	4
format	foglio4	3	3	4

Figura 4: Struttura della tabella format

Come è possibile notare dalla figura 4, questa è una tabella di tipo *bag* popolata da record di tipo:
$-record(format, \{foglio, tab\_index, nrighe, ncolonne\})$.
In particolare:

- Nel campo $foglio$ verrà inserito il nome del foglio di cui si vuole memorizzare la struttura
- Nel campo *tab_index* verrà inserito l'indice della tabella di un determinato foglio
- Nel campo *nrighe* verrà inserito il numero di righe della tabella specificata dal campo *tab_index*
- Nel campo *ncolonne* verrà inserito il numero di colonne della tabella specificata dal campo *tab_index*

Dettagli implementativi

Per quanto riguarda l'implementazione dell'applicazione in linguaggio Erlang si riporta la descrizione di una funzione rilevante per ogni modulo. Il codice completo è reperibile al repository GitHub pubblico: https:// github.com/ GiacomoDiFa/ Progetto-ADCC.

- La funzione *new* del modulo *spreadsheet* crea su tutti i nodi della rete un nuovo foglio (implementato come una tabella Mnesia), se non già presente, e popola tale foglio col valore *undef*. Infine salva il Pid del nodo proprietario del foglio che ha invocato la funzione e il formato del foglio (numero di righe e colonne per ogni sottotabella).

```
new(TabName, N, M, K) ->
    mnesia:start(),
    TabelleLocali = mnesia:system_info(tables),
    case lists:member(TabName, TabelleLocali) of
        true -> {error, invalid_name};
        false ->
            SpreadsheetFields = record_info(fields, spreadsheet),
            NodeList = [node()]++nodes(),
            mnesia:create_table(TabName, [
                {attributes, SpreadsheetFields},
                {disc_copies, NodeList},
                {type, bag}
            ]),
            % popolo il foglio con k tabelle di n righe e m colonne
            popola_foglio(TabName, K, N, M),
            % il nodo è proprietario del foglio
            popola_owner_table(TabName),
            % salvo le informazioni per il formato della tabella
            popola_format_table(TabName, K, N, M)
    end
.
```

- La funzione *create_table_distrib* del modulo *distribution* crea una volta per tutte le tabelle *owner*, *policy* e $format$, già illustrate in precedenza, in tutti i nodi della rete. Prima di terminare interrompe il DBMS Mnesia che deve essere fatto ripartire se si vuole usare il DB (tramite il metodo *distribution:start_distrib*).

```
create_table_distrib() ->
    % creo il DB in locale -> node() e in remoto -> nodes()
    NodeList = [node()] ++ nodes(),
    mnesia:create_schema(NodeList),
    % faccio partire Mnesia nei nodi remoti e da me
    start_remote(),
    OwnerFields = record_info(fields, owner),
    PolicyFields = record_info(fields, policy),
    FormatFields = record_info(fields, format),
    mnesia:create_table(owner, [
        {attributes, OwnerFields},
        {disc_copies, NodeList}
    ]),
    mnesia:create_table(policy, [
```

```
            {attributes, PolicyFields},
            {disc_copies, NodeList},
            {type, bag}
        ]),
        mnesia:create_table(format, [
            {attributes, FormatFields},
            {disc_copies, NodeList},
            {type, bag}
        ]),
        % stop per ogni nodo remoto e per me
        distribution:stop_distrib()
    .
```

- La funzione *to_csv* del modulo *csv_manager* permette di salvare su disco (nel nodo che invoca la funzione) un foglio (tabella Mnesia) in formato CSV.

```
    to_csv(TableName, FileName) ->
        mnesia:start(),
        TabelleLocali = mnesia:system_info(tables),
        case lists:member(TableName, TabelleLocali) of
            false -> {error, invalid_name_of_table};
            true ->
                % Apri il file per la scrittura
                {ok, File} = file:open(FileName, [write]),
                % Estrai i dati dalla tabella Mnesia
                Records = ets:tab2list(TableName),
                % Converti i dati in formato CSV
                CsvContent = records_to_csv(Records),
                % Scrivi il CSV nel file
                file:write(File, CsvContent),
                % Chiudi il file
                file:close(File)
        end
    .
```

- La funzione *average_lookup_time* del modulo *debug* permette di effettuare un numero di N letture ad una cella di un foglio specifico (Foglio) per poter calcolare il tempo medio di accesso in lettura al foglio (calcolato in microsecondi).

```
    % media aritmetica
    average_lookup_time(Foglio, N) ->
        Times = lists:map(fun(_I) ->
                {_TimeUnit, Time} = lookup_time(Foglio),
                Time
            end,
            lists:seq(1, N)
        ),
        SumOfTimes = lists:sum(Times),
        Length = lists:foldl(fun(_X, Acc) -> Acc + 1 end, 0, Times),
        {microsecond, SumOfTimes / Length}
```

```
.

lookup_time(Foglio) ->
    TimeUnit = microsecond,
    {Time, _Value} = timer:tc(spreadsheet, get, [Foglio, 1, 1, 1], TimeUnit),
    {TimeUnit, Time}
.
```

Funzionamento nel contesto distribuito

Per utilizzare l'applicazione nel contesto distribuito occorre seguire i seguenti passi:

1. Per ogni nodo della rete che farà parte del sistema distribuito, eseguire il comando:
   erl -sname "*nome_del_nodo*" -setcookie "*nome_del_cookie*"
   in modo da attivare la macchina virtuale Erlang su tale nodo (ogni nodo avrà un nome diverso ma tutti condivideranno lo stesso cookie).

2. Compilare i moduli con i comandi (per ogni nodo):

   - c(spreadsheet).
   - c(distribution).
   - c(csv_manager).
   - c(debug).

3. A questo punto, per comunicare con gli altri nodi presenti nella rete, risulta necessario pingare ogni nodo. É sufficiente che un nodo contatti tutti gli altri. Per fare ciò basta usare il metodo:

   - net_adm:ping(nome_del_nodo).

4. Successivamente, per il corretto funzionamento del programma nel contesto distribuito, è necessario salvare globalmente il nome di ogni nodo presente nella rete con il metodo (da eseguire internamente ad ogni nodo):

   - global:register_name(nome_del_nodo, self()).

   In questo modo, sarà sempre possibile ottenere il Pid globale di ciascun nodo con il metodo (utile per il corretto funzionamento della funzione *spreadsheet:share*):

   - global:whereis_name(nome_del_nodo).

5. Per eseguire il programma in maniera distribuita è possibile utilizzare i seguenti metodi (su un qualsiasi nodo facente parte della rete):

   - distribution:create_table_distrib().
     Per creare in modo distribuito le tabelle Mnesia su tutti i nodi della rete.
   - distribution:start_distrib().
     Per avviare il DBMS Mnesia su tutti i nodi della rete.

6. A questo punto, è possibile impiegare tutti i metodi precedentemente illustrati per creare, condividere e lavorare sui vari fogli.

7. Per interrompere l'esecuzione nel contesto distribuito (su tutti i nodi), è sufficiente eseguire il seguente metodo (su un qualsiasi nodo facente parte della rete):

   - distribution:stop_distrib().

Misurazioni

Le misurazioni e i test sono stati realizzati in locale tramite il modulo *debug*, in particolare usando le funzioni *test_sequential*, *test_mixed*, *test_battery*, *average_lookup_time* e *average_setup_time*. I test effettuati sono i seguenti:

- Test sequenziale: creo 5 processi (lettori) e per ognuno di essi effettuo una lettura su uno stesso foglio. Poi creo altri 5 processi (scrittori) e per ognuno di essi effettuo una scrittura sempre sullo stesso foglio.
  - Metodo di invocazione del test:
    debug:test_sequential(5, foglio1).
  - Risultati del test:

```
{{readers,[<0.209.0>,<0.210.0>,<0.211.0>,<0.212.0>,<0.213.0>],
          [undef,undef,undef,undef,undef]},
 {writers,[<0.219.0>,<0.220.0>,<0.221.0>,<0.222.0>,<0.223.0>],
          [ok,ok,ok,ok,ok]}}
```

- Test misto: effettuo un test sequenziale su 5 processi (5 lettori e 5 scrittori) ripetendolo per 3 volte.
  - Metodo di invocazione del test:
    debug:test_mixed(5, 3, foglio1).
  - Risultati del test:

```
{{readers,[<0.270.0>,<0.271.0>,<0.272.0>,<0.273.0>,<0.274.0>]},
 {writers,[<0.275.0>,<0.276.0>,<0.277.0>,<0.278.0>,<0.279.0>]},
 {results,[
   {{test,1},
    ['WRITE','WRITE','WRITE','WRITE','WRITE'],
    [ok,ok,ok,ok,ok]},
   {{test,2},
    ['WRITE','WRITE','WRITE','WRITE','WRITE'],
    [ok,ok,ok,ok,ok]},
   {{test,3},
    ['WRITE','WRITE','WRITE','WRITE','WRITE'],
    [ok,ok,ok,ok,ok]}]}}
```

- Test a batteria:
  - il test è costituito da 3 sotto-test:
    a. Test 1: test sequenziale con 100 processi. Tempo di esecuzione: circa 390 ms (0,390 sec).
    b. Test 2: test misto con 100 processi e 10 operazioni di lettura-scrittura. Tempo di esecuzione: circa 2819 ms (2,819 sec).
    c. Test 3: test misto (stress) con 1000 processi e 10 operazioni di lettura-scrittura. Tempo di esecuzione: circa 15797 ms (15,797 sec).
  - Metodo di invocazione del test:
    debug:test_battery(foglio1).
  - Risultati del test:

```
{{sequential_test,{params,[100,foglio1]},{time_ms,390}},
 {mixed_test,{params,[100,10,foglio1]},{time_ms,2819}},
 {stress_mixed_test,{params,[1000,10,foglio1]},{time_ms,15797}}}
```

- Test di lettura: calcolo il tempo medio di una lettura su un numero variabile di misurazioni (N).
  - Metodo di invocazione del test:
    debug:average_lookup_time(foglio1, N).
  - Risultati del test:
    - Test con N = 10: {microsecond, 532.0}
    - Test con N = 100: {microsecond, 496.39}
    - Test con N = 1000: {microsecond, 434.578}
    - Test con N = 10000: {microsecond, 370.5683}
    - Test con N = 100000: {microsecond, 345.66007}
- Test di scrittura: calcolo il tempo medio di una scrittura su un numero variabile di misurazioni (N).
  - Metodo di invocazione del test:
    debug:average_setup_time(foglio1, N).
  - Risultati del test:
    - Test con N = 10: {microsecond, 593.8}
    - Test con N = 100: {microsecond, 576.04}
    - Test con N = 1000: {microsecond, 548.386}
    - Test con N = 10000: {microsecond, 519.5912}
    - Test con N = 100000: {microsecond, 483.85945}