1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

UNIVERSITÀ DEGLI STUDI DI URBINO CARLO BO
DIPARTIMENTO DI SCIENZE PURE E APPLICATE
CORSO DI LAUREA DI INFORMATICA E INNOVAZIONE DIGITALE

Progetto di Applicazioni Distribuite e Cloud Computing

Sessione Invernale 2024/2025

Prof. Claudio Antares Mezzina

Marzio Della Bosca
Matricola: 329608

Susanna Peretti
Matricola: 329456

Indice

1
Specifica del Problema

Il progetto consiste nell'implementazione di uno spazio delle tuple (Tuple Space), un'astrazione di memoria condivisa che consente la comunicazione tra processi concorrenti tramite la condivisione di tuple. Lo spazio delle tuple fornisce un ambiente in cui i processi possono leggere e scrivere dati in modo asincrono e indipendente, utilizzando un'interfaccia definita che consente le operazioni fondamentali di inserimento, lettura e rimozione di tuple.

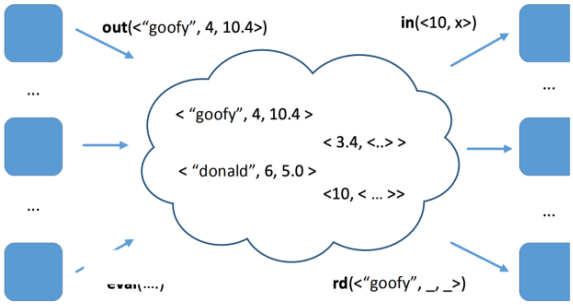

Figura 1.1: Tuple Space

2
Analisi del Problema

Le operazioni di base fornite dall'interfaccia sono le seguenti:

`new/1`: Questa operazione crea un nuovo spazio delle tuple con un nome specificato. Ciò implica la necessità di un sistema in grado di generare uno spazio di memoria in cui le tuple possano essere conservate e accessibili. Ogni spazio delle tuple creato sarà indipendente e avrà una propria istanza di dati associata.

`out/2`: Inserisce una tupla nello spazio delle tuple. Questa è l'unica operazione di scrittura.

`in/2`: Esegue una lettura e rimozione della tupla che corrisponde al pattern specificato. Questa operazione implica che un nodo potrebbe essere bloccato in attesa della disponibilità di una tupla che soddisfi il pattern richiesto.

`rd/2`: Simile all'operazione `in/2`, ma la tupla letta non viene rimossa. Questo permette ai nodi di *leggere* i dati senza alterare lo stato dello spazio delle tuple, evitando modifiche non desiderate.

Le versioni con timeout di `in/3` e `rd/3` sono introdotte per gestire i casi in cui una tupla che soddisfa il pattern richiesto non sia immediatamente disponibile. Queste versioni permettono di evitare il blocco indefinito dei processi, restituendo un errore se il tempo di attesa supera un valore definito, rendendo l'operazione più resiliente a situazioni di stallo.

Infine, le operazioni di gestione dei nodi sono:

`addNode/2`: Aggiunge un nodo allo spazio condiviso, consentendo a un nuovo processo di interagire con il sistema e di poter leggere e scrivere tuple.

`removeNode/2`: Rimuove un nodo dallo spazio delle tuple, disabilitando il suo accesso alle tuple. Se un nodo non è più in grado di comunicare o se deve essere disconnesso, questa operazione lo rimuove dal sistema senza compromettere l'integrità dello spazio delle tuple.

`nodes/1`: Restituire un elenco dei nodi attivi nello spazio delle tuple è importante per consentire la gestione della rete di nodi, ad esempio per operazioni di bilanciamento del carico, monitoraggio dello stato del sistema, e per facilitare operazioni di debug.

3
Scelte di Progetto

Il progetto è strutturato in tre file distinti, ciascuno con una responsabilità specifica:

- `tuples_space.erl`: modulo principale che gestisce lo spazio condiviso. Esso definisce l'interfaccia per la creazione di tuple e la registrazione dei nodi.
- `nodo_ts.erl`: modulo che definisce l'interfaccia dei nodi e le modalità di comunicazione con lo spazio delle tuple. Abbiamo scelto di separare questi moduli per riflettere meglio l'architettura di un sistema distribuito basato su un modello client-server.
- `test.erl`: modulo dedicato all'esecuzione di test per verificare il corretto funzionamento dell'applicazione.

I moduli `tuples_space.erl` e `nodo_ts.erl`, implementano un loop interno per gestire i dati e mantenere le istanze *reattive*. Per aumentare la resilienza del sistema, è stata introdotta una soluzione di recovery basata sul pattern *watchdog*. In questo contesto, se la funzione loop dovesse cessare di funzionare per qualche motivo, la funzione di guardia interviene e avvia una nuova istanza del loop, garantendo così la continuità operativa del sistema.

Quindi, lo spazio delle tuple viene mantenuto attivo tramite un processo *guardiano*, che monitora la sua esecuzione.

La gestione dei timeout è stata definita nelle interfacce anziché essere implementata direttamente nel loop di esecuzione del modulo gestore. Questa scelta ottimizza il codice, permettendo alle varianti delle funzioni di interfaccia che prevedono un timeout, come `in` e `rd`, di richiamare la stessa funzione interna, con il timeout gestito localmente.

Le funzioni principali per la ricerca delle tuple e la validazione dell'autorizzazione dei nodi sono:

- `filter`: utilizza la funzione `member` per verificare la presenza di una tupla, rimuovendola e restituendo il resto della lista.
- `matching_nodes`: verifica che un nodo sia registrato correttamente allo spazio delle tuple, impiega anch'essa la funzione `member`.

I test sono progettati per misurare le prestazioni dell'applicazione e valutare la sua resilienza. A tale scopo, utilizziamo la funzione `timer:tc`, che permette di misurare il tempo di esecuzione delle operazioni testate. Sono stati implementati tre tipi di test:

- Misurazione dei tempi di esecuzione nel modulo gestore dello spazio delle tuple;
- Misurazione dei tempi di esecuzione nel modulo gestore dei nodi;
- Test di resilienza e recupero in caso di crash.

Considerando la natura casuale di un sistema distribuito, per i primi due test è possibile eseguire più iterazioni della stessa operazione e calcolare la media dei tempi di esecuzione. Inoltre, a causa della nostra implementazione, gli spazi delle tuple tendono a riempirsi progressivamente durante l'esecuzione automatica dei test. Di conseguenza, ci aspettiamo un incremento dei tempi medi di esecuzione all'aumentare del numero di iterazioni richieste.

4
Implementazione

4.1 Codice Gestore - `tuples_space`

La funzione `new/1` crea un nuovo Tuple Space. Lancia un processo (il "watchdog") che monitora lo stato e la salute dello spazio delle tuple. Ogni spazio delle tuple è identificato da un nome e un processo di supervisione. La funzione `watchdog/2` gestisce un ciclo di vita del Tuple Space. Controlla il processo di "loop" (che gestisce le operazioni del Tuple Space) e riavvia se dovesse terminare inaspettatamente.

```
%%% Creazione di un nuovo Tuple Space
new(Name) ->
    Mio = self(),
    Pid = spawn(fun() -> watchdog(Name, [Mio]) end),
    {ok, Pid}.

%%% Watchdog che avvia e monitora il loop
watchdog(Name, Nodes) ->
    % Permette di catturare l'uscita del processo figlio
    process_flag(trap_exit, true),
    LoopPid = spawn_link(?MODULE, loop, [{[], Nodes, self()}]),
    register(Name, LoopPid),
    receive
        {'EXIT', LoopPid, Reason} ->
            io:format("Loop terminato con motivo: ~p. Riavvio...~n", [Reason]),
            watchdog(Name, Nodes);
        {crash_test, From} ->
            From ! {ok, restarted},
            watchdog(Name, Nodes)
    end.
```

4.1.1 Funzioni interfaccia

```
in(Pid, Tuple) ->
    Pid ! {in, self(), Tuple},
    receive
        {ok, Tuple} ->
            {ok, Tuple};
        {err, unfollowed} ->
            {err, unfollowed}
    end.
```

```
in(Pid, Tuple, Timeout) ->
    Pid ! {in, self(), Tuple},
    receive
        {ok, Tuple} ->
            {ok, Tuple};
        {err, unfollowed} ->
            {err, Pid, unfollowed}
    after Timeout * 1000 ->
        {timeout, [Timeout]}
    end.

rd(Pid, Tuple) ->
    Pid ! {rd, self(), Tuple},
    receive
        {ok, Tuple} ->
            {ok, Tuple};
        {err, unfollowed} ->
            {err, self(), Pid, unfollowed}
    end.

rd(Pid, Tuple, Timeout) ->
    Pid ! {rd, self(), Tuple},
    receive
        {ok, Tuple} ->
            {ok, Tuple};
        {err, unfollowed} ->
            {err, self(), Pid, unfollowed}
    after Timeout * 1000 ->
        {timeout, [Timeout]}
    end.

out(Pid, Tuple) ->
    Pid ! {out, self(), Tuple},
    receive
        {ok, Tuple} -> {ok, Tuple};
        {err, unfollowed} -> {err, self(), Pid, unfollowed}
    end.

addNode(Pid, Node) ->
    Pid ! {addNode, Node, self()},
    receive
        {ok, Node} -> {ok, Node, Pid};
        {err, followed} -> {err, Pid, already_followed}
    end.

removeNode(Pid, Node) ->
    Pid ! {removeNode, Node, self()},
    receive
        {ok, Node} -> {ok, Node, Pid};
        {err, unfollowed} -> {err, Pid, unfollowed}
    end.
```

```
nodes(Pid) ->
    Pid ! {nodes, self()},
    receive
        {ok, Nodes} -> Nodes;
        {err, unfollowed} -> {err, Pid, unfollowed}
    end.
```

4.1.2 Funzione server

Processo centrale che riceve e gestisce i messaggi provenienti dai nodi connessi, utilizza le funzioni di validazione dei nodi e delle tuple.

```
loop({Tuples, Nodes, WatchdogPid} = State) ->
    receive
        {in, From, Tuple} ->
            case matching_nodes(From, Nodes) of
                % se il nodo non è registrato a questo spazio tuple non
                può eseguire operazioni
                {err, From} ->
                    From ! {err, unfollowed},
                    loop(State);
                {ok, From} ->
                    case filter(Tuple, Tuples) of
                        % ritorna la tupla se è presente nello spazio, in questo caso
                        è anche rimossa
                        {err, Tuple} ->
                            From ! {err, Tuple},
                            loop(State);
                        {ok, Tuple, Rest} ->
                            From ! {ok, Tuple},
                            loop({Rest, Nodes, WatchdogPid})
                    end
            end;
        {rd, From, Tuple} ->
            case matching_nodes(From, Nodes) of
                {err, From} ->
                    From ! {err, unfollowed},
                    loop(State);
                {ok, From} ->
                    case filter(Tuple, Tuples) of
                        {err, Tuple} ->
                            From ! {err, Tuple},
                            loop(State);
                        {ok, Tuple, _} ->
                            From ! {ok, Tuple},
                            loop(State)
                    end
            end;
        {out, From, Tuple} ->
```

```
            case matching_nodes(From, Nodes) of
                {err, From} ->
                    From ! {err, unfollowed},
                    loop(State);
                {ok, From} ->
                    From ! {ok, Tuple},
                    loop({[Tuple | Tuples], Nodes, WatchdogPid})
            end;
        {addNode, Node, From} ->
            case matching_nodes(Node, Nodes) of
                {err, Node} ->
                    From ! {ok, Node},
                    loop({Tuples, [Node | Nodes], WatchdogPid});
                {ok, Node} ->
                    From ! {err, followed},
                    loop(State)
            end;
        {removeNode, Node, From} ->
            case matching_nodes(Node, Nodes) of
                {err, Node} ->
                    From ! {err, unfollowed},
                    loop(State);
                {ok, Node} ->
                    From ! {ok, Node},
                    Rest = lists:delete(Node, Nodes),
                    loop({Tuples, Rest, WatchdogPid})
            end;
        {nodes, Node} ->
            case matching_nodes(Node, Nodes) of
                {err, Node} ->
                    Node ! {err, unfollowed},
                    loop(State);
                {ok, Node} ->
                    Node ! {ok, Nodes},
                    loop(State)
            end;
        {crash_test, From} ->
            WatchdogPid ! {crash_test, From},
            break
    end.
```

4.1.3 Funzioni ausiliarie

`filter/2`: Verifica se una tupla è presente nello spazio delle tuple e, in caso affermativo, la rimuove. Se la tupla non è presente, restituisce un errore. `matching_nodes/2`: Verifica se un nodo (processo) è registrato nel Tuple Space. Se sì, permette l'esecuzione dell'operazione; altrimenti, restituisce un errore.

```
filter(Tupla, Tuples) ->
    case lists:member(Tupla, Tuples) of
        false ->
            {err, Tupla};
```

```
        true ->
            {ok, Tupla, lists:delete(Tupla, Tuples)}
    end.

matching_nodes(Pid, Nodes) ->
    case lists:member(Pid, Nodes) of
        false ->
            {err, Pid};
        true ->
            {ok, Pid}
    end.
```

5
Testing

5.1 Test Funzionali

Modulo `tuples_space`

```
Eshell V15.1.2 (press Ctrl+G to abort, type help(). for help)
1> c(tuples_space).
{ok,tuples_space}
2> tuples_space:new(t).
{ok,<0.91.0>}
3> tuples_space:out(t,{hello,world}).
{ok,{hello,world}}
4> tuples_space:rd(t,{hello,world}).
{ok,{hello,world}}
5> tuples_space:in(t,{hello,world}).
{ok,{hello,world}}
6> tuples_space:rd(t,{hello,world},2).
{timeout,[2]}
7> tuples_space:in(t,{hello,world},2).
{timeout,[2]}
```

Figura 5.1: `out/2` → `rd/2` → `in/2` → `rd/3` → `in/3`

```
Eshell V15.1.2 (press Ctrl+G to abort, type help(). for help)
1> c(tuples_space),c(nodo_ts).
{ok,nodo_ts}
2> tuples_space:new(t).
{ok,<0.95.0>}
3> tuples_space:nodes(t).
[<0.84.0>]
4> nodo_ts:init(uno).
{ok,<0.99.0>}
5> tuples_space:addNode(t,uno).
{ok,uno,t}
6> tuples_space:nodes(t).
[uno,<0.84.0>]
7> tuples_space:removeNode(t,uno).
{ok,uno,t}
8> tuples_space:nodes(t).
[<0.84.0>]
```

Figura 5.2: `addNode/2` → `nodes/1` → `removeNode/2` → `nodes/1`

```
Eshell V15.1.2 (press Ctrl+G to abort, type help(). for help)
1> c(tuples_space).
{ok,tuples_space}
2> tuples_space:new(t).
{ok,<0.91.0>}
3> tuples_space:out(t,{hello,world}).
{ok,{hello,world}}
4> tuples_space:rd(t,{hello,world}).
{ok,{hello,world}}
5> tuples_space:crash_test(t).
{ok,restarted}
6> tuples_space:rd(t,{hello,world},2).
{timeout,[2]}
7> tuples_space:out(t,{hello,world}).
{ok,{hello,world}}
8> tuples_space:rd(t,{hello,world}).
{ok,{hello,world}}
```

Figura 5.3: Recovery test

Modulo `node_ts`

```
Eshell V15.1.2 (press Ctrl+G to abort, type help(). for help)
1> c(tuples_space),c(nodo_ts).
{ok,nodo_ts}
2> nodo_ts:init(uno).
{ok,<0.95.0>}
3> tuples_space:new(t).
{ok,<0.97.0>}
4> tuples_space:addNode(t,uno).
{ok,uno,t}
5> nodo_ts:out(uno,t,{hello,world}).
{ok,{hello,world}}
6> nodo_ts:rd(uno,t,{hello,world}).
{ok,{hello,world}}
7> nodo_ts:in(uno,t,{hello,world}).
{ok,{hello,world}}
8> nodo_ts:rd(uno,t,{hello,world},2).
{timeout,[2]}
9> nodo_ts:in(uno,t,{hello,world},2).
{timeout,[2]}
```

Figura 5.4: `out/2` → `rd/2` → `in/2` → `rd/3` → `in/3`

5.2 Validazioni

```
Eshell V15.1.2 (press Ctrl+G to abort, type help(). for help)
1> c(tuples_space),c(nodo_ts).
{ok,nodo_ts}
2> tuples_space:new(t).
{ok,<0.95.0>}
3> nodo_ts:init(uno),nodo_ts:init(due).
{ok,<0.99.0>}
4> tuples_space:addNode(t,uno).
{ok,uno,t}
5> tuples_space:nodes(t).
[uno,<0.84.0>]
6> nodo_ts:out(uno,t,{hello,world}).
{ok,{hello,world}}
7> nodo_ts:rd(due,t,{hello,world}).
{error,t,unfollowed}
8> tuples_space:nodes(t).
[uno,<0.84.0>]
```

Figura 5.5: `out/2`, `rd/2`

```
Eshell V15.1.2 (press Ctrl+G to abort, type help(). for help)
1> c(tuples_space).
{ok,tuples_space}
2> tuples_space:new(t).
{ok,<0.91.0>}
3> tuples_space:nodes(t).
[<0.84.0>]
4> tuples_space:addNode(t,self()).
{err,t,already_followed}
5> c(nodo_ts).
{ok,nodo_ts}
6> nodo_ts:init(uno).
{ok,<0.101.0>}
7> tuples_space:removeNode(t,uno).
{err,t,unfollowed}
```

Figura 5.6: `addNode/2`, `removeNode/2`

5.3 Test di Performance

```
Eshell V15.1.2 (press Ctrl+G to abort, type help(). for help)
1> test:run_tests(1000).

Testing average execution times, (1000 run)...

Average execution times from main module(microseconds):
in: 1.836, rd: 0.918, out: 1.734, addNode: 4.08, removeNode: 1.734

Average execution times from node module(microseconds):
in: 5.406, rd: 4.692, out: 4.386
Recovery time: 0 microseconds
ok
```

Figura 5.7: Media su 1000 run