Università degli Studi di Urbino Carlo Bo
Corso di Laurea Magistrale in Informatica Applicata

Applicazioni Distribuite e Cloud Computing

Implementazione di una Tuple Space Distribuita in Erlang

Docente: Prof. Claudio Antares Mezzina
Studente: Brahim El Hannaoui
Matricola: 315597
Anno Accademico: 2024–2025

04/09/2025

Indice

Capitolo 1

Introduzione

L'obiettivo di questo progetto è costruire un sistema di **Tuple Space (TS)** minimale in Erlang. Questo progetto serve a consolidare i concetti fondamentali delle applicazioni distribuite in Erlang, tra cui: processi, messaggi, pattern matching, gestione dei timeout e interazione tra più nodi. L'idea è semplice: il TS è una lavagna condivisa.

1.1 Fasi di Implementazione (Tre Step)

La costruzione del sistema avviene in tre fasi distinte, che aggiungono funzionalità passo dopo passo.

Interfaccia 1/3 - API Base

In questa prima fase, l'obiettivo è implementare le funzionalità di base del Tuple Space. Si crea una nuova istanza del Tuple Space tramite `new/1`, che registra un processo server con un nome specifico. Le operazioni principali di interazione sono `out/2` per aggiungere una tupla, `rd/2` per leggerla senza rimuoverla e `in/2` per leggerla e rimuoverla.

Interfaccia 2/3 - Gestione dei Timeout

Nella seconda fase, si migliora l'esperienza del client introducendo la gestione dei timeout. Le nuove funzioni `rd/3` e `in/3` permettono al client di specificare un tempo massimo di attesa. Se la tupla viene trovata entro il tempo limite, la funzione restituisce `{ok, Tuple}`; altrimenti, ritorna `{err, timeout}`. Questa scelta mantiene il server del Tuple Space più snello, delegando la logica di riprova e attesa al client.

Interfaccia 3/3 - Sistema Multi-Nodo

L'ultima fase estende il sistema per supportare più nodi, ma senza implementare la replica dei dati. Questo viene realizzato tramite un processo **proxy**. Il proxy, che si trova sui nodi remoti, inoltra le richieste al server master. Le funzioni `addNode`, `removeNode` e `nodes` permettono di gestire l'elenco dei nodi accessibili e monitorarne lo stato.

—

Semplificazioni

- solo **flat tuples**;
- **timeout** e blocchi implementati sul *client* (`receive ... after`);
- un solo TS **master** (nessuna replica dei dati).

Capitolo 2

Design decisions

2.1 Stato centralizzato e loop

Un **unico processo server** che contiene tutte le tuple: il TS "vero" vive su un nodo master. Così `in/2` (che consuma) resta naturalmente atomico: c'è un solo punto che toglie la tupla. Il codice è lineare e il debug semplice.

Stato minimale: Tuples (lista di tuple) e Proxies (lista di Node,Pid dei proxy). Il server riceve messaggi e richiama loop(...) con lo stato aggiornato.

```
loop(Tuples, Proxies) ->
  receive
    {out, T} ->
      loop([T | Tuples], Proxies);

    {rd, From, P} ->
      case find_match(P, Tuples) of
        {ok, M} -> From ! {ok, M}, loop(Tuples, Proxies);
        not_found -> From ! not_found, loop(Tuples, Proxies)
      end;

    {in, From, P} ->
      case take_match(P, Tuples) of
        {ok, M, Rest} -> From ! {ok, M}, loop(Rest, Proxies);
        not_found     -> From ! not_found, loop(Tuples, Proxies)
      end

    %% (qui: add_node/remove_node/nodes + {nodedown,Node})
  end.
```

2.2 Timeout lato client

`rd` e `in` possono non trovare subito. Per non complicare il server con code d'attesa, metto il **"aspetta e riprova"** sul client: il server risponde sempre `{ok,Tuple}` oppure `not_found`; se il client ha ancora *budget*, aspetta uno *step* e ripete. Comportamento prevedibile, server pulito.
Schema essenziale (solo `rd`; `in` è analogo)

```
rd(TS, P, Timeout) -> rd_until(TS, P, Timeout).

rd_until(_TS, _P, 0) -> {err, timeout};
rd_until(TS,  P,  T) ->
  TS ! {rd, self(), P},
  receive
    {ok, M}   -> {ok, M};
    not_found ->
      Step = if T >= 10 -> 10; true -> T end,
      receive after Step ->
        New = T - Step,
        if New =< 0 -> {err, timeout}; true -> rd_until(TS, P,
           New) end
      end
  end.
```

Note che contano: `not_found` = "adesso non c'è"; `timeout` = "non trovato entro il tempo". Lo *step* è $\min(10ms, \texttt{Timeout})$.

2.3 Ricerca con e senza consumo (idea)

- `find_match` scorre la lista e ritorna la prima tupla che combacia;
- `take_match` fa lo stesso ma costruisce anche la lista senza quella tupla, così `in/2` può consumarla aggiornando lo stato a `Rest`.

2.4 Pattern matching semplice (flat tuples, jolly '_')

Le tuple devono avere **stessa arità**; poi confronto campo-per-campo con un jolly chiaro: l'atomo '_'.

Cuore del matching (estratto)

```
matches(Pat, Tup)
  when is_tuple(Pat), is_tuple(Tup),
       tuple_size(Pat) =:= tuple_size(Tup) ->
  match_lists(tuple_to_list(Pat), tuple_to_list(Tup));
matches(_, _) -> false.

match_field('_', _)      -> true;    % jolly
match_field(Value, Value) -> true;
match_field(_, _)        -> false.
```

2.5 Visibilità multi-nodo tramite proxy

Voglio usare il nome locale `ts1` anche su nodi remoti **senza replicare i dati**. Creo su quel nodo un *proxy* registrato `ts1` che **inoltra** i messaggi al master. Così su A posso invocare `ts_step3:rd(ts1, ...)` e il proxy inoltra al TS su B.

Proxy essenziale

```erlang
start_proxy(Name, MasterPid) ->
  case whereis(Name) of
    undefined ->
      Pid = spawn(fun() -> register(Name,self()),
                           proxy_loop(MasterPid) end),
      {ok, Pid};
    _ -> {error, name_in_use}
  end.

proxy_loop(MasterPid) ->
  receive
    stop -> ok;
    Msg  -> MasterPid ! Msg, proxy_loop(MasterPid)
  end.
```

2.6 Auto-pulizia dei nodi caduti

Quando un nodo con proxy scende, voglio che `nodes/1` si aggiorni da solo. Sul master attivo `erlang:monitor_node(Node,true)` al `addNode`; se arriva `{nodedown,Node}`, rimuovo quel nodo dalla lista dei proxy.

```erlang
{nodedown, Node} ->
  case lists:keytake(Node, 1, Proxies) of
    {value, {_N,_Pid}, New} -> loop(Tuples, New);
    false                   -> loop(Tuples, Proxies)
  end.
```

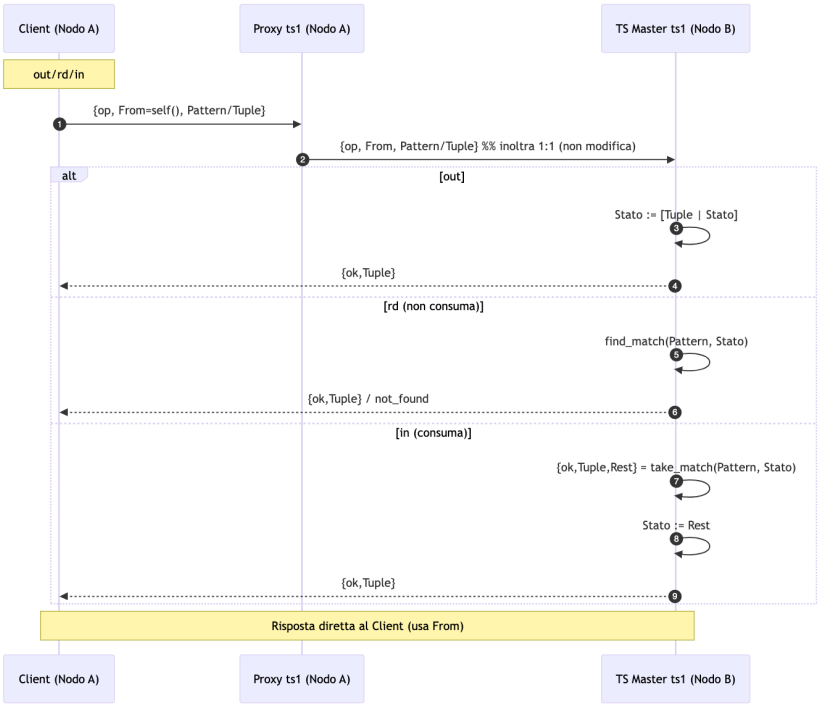

Figura 2.1: Architettura generale del Tuple Space

Capitolo 3

Test e validazioni

Questa sezione documenta i test eseguiti *interamente da shell Erlang.*

Setup nodi

```
%% Nodo B (master)
%% ERL_AFLAGS="-sname b1 -setcookie choco" rebar3 shell

%% Nodo A (client)
%% ERL_AFLAGS="-sname a1 -setcookie choco" rebar3 shell
```

3.1 Test locale — sul nodo master (B)

```
(b1@brahims-MacBook-Pro)1> {ok,_} = ts_step3:new(ts1).
{ok,ts1}
(b1@brahims-MacBook-Pro)2> ts_step3:out(ts1, {x,1}).
ok
(b1@brahims-MacBook-Pro)3> ts_step3:rd(ts1, {x,'_'}).
{x,1}
(b1@brahims-MacBook-Pro)4> ts_step3:rd(ts1, {x,'_'}).
{x,1}
(b1@brahims-MacBook-Pro)5> ts_step3:in(ts1, {x,'_'}).
{x,1}
(b1@brahims-MacBook-Pro)6> ts_step3:out(ts1, {u,2}).
ok
(b1@brahims-MacBook-Pro)7> ts_step3:rd(ts1, {'_',2}).
{u,2}
(b1@brahims-MacBook-Pro)8> ts_step3:rd(ts1, {u,'_','_'}, 10).
{err,timeout}
```

3.2 Test con proxy (visibilità del nome su A)

```
%% su B: crea proxy su A e verifica visibilit
```

```
(b1@brahims-MacBook-Pro)9> ts_step3:addNode(ts1, 'a1@brahims-
   MacBook-Pro').
ok
(b1@brahims-MacBook-Pro)10>  ts_step3:nodes(ts1).
['b1@brahims-MacBook-Pro','a1@brahims-MacBook-Pro']

%% ora su A posso usare "ts1" (nome locale)
(a1@brahims-MacBook-Pro)2>  ts_step3:out(ts1, {z,3}).
ok
(a1@brahims-MacBook-Pro)3> ts_step3:rd(ts1, {z,'_'}).
{z,3}
```

3.3 Timeout lato client (polling)

```
%% su A: nessuno ha messo {w,_} -> scade
(a1@brahims-MacBook-Pro)1> ts_step3:rd({ts1,'b1@brahims-MacBook-
   Pro'}, {t,'_'}, 500).
{err,timeout}

%% scenario positivo:
(a1@brahims-MacBook-Pro)4> ts_step3:rd({ts1,'b1@brahims-MacBook-
   Pro'}, {t,'_'}, 1500). %% A resta in attesa
%% su B:
(b1@brahims-MacBook-Pro)11> ts_step3:out(ts1, {t,1}).
ok
%% su A: {ok,{t,1}}
```

3.4 Performance (medie in μs)

```
%% Locale (su B)
(b1@brahims-MacBook-Pro)48> ts_runner:run_local(1000).
LOCALE (n=1000)
out/2 = 0.383 us
rd/2  = 1.127 us
in/2  = 1.166 us
ok

%% Remoto senza proxy (da A verso B, con timeout)
(a1@brahims-MacBook-Pro)1> ts_runner:run_remote(1000, 'b1@brahims
   -MacBook-Pro', 500).
REMOTO A->B (n=1000, timeout=500 ms)
out/2 = 1.286 us
rd/3  = 70.338 us
in/3  = 47.665 us
ok
```

3.5 Recovery (tempo percepito lato API, ms)

```
(b1@brahims-MacBook-Pro)44> net_adm:ping('a1@brahims-MacBook-Pro').
pong
(b1@brahims-MacBook-Pro)45> ts_step3:addNode(ts1, 'a1@brahims-MacBook-Pro').
ok
(b1@brahims-MacBook-Pro)46> lists:member('a1@brahims-MacBook-Pro', ts_step3:nodes(ts1)).
true
(b1@brahims-MacBook-Pro)47> RT1 = ts_runner:recovery_ms(ts1,'a1@brahims-MacBook-Pro', 10),
                            io:format("Recovery = ~p ms~n", [RT1]).
Recovery = 11 ms
ok
(b1@brahims-MacBook-Pro)48> ts_step3:addNode(ts1, 'a1@brahims-MacBook-Pro').
ok
(b1@brahims-MacBook-Pro)49> RT2 = ts_runner:recovery_ms(ts1,'a1@brahims-MacBook-Pro', 10),
                            io:format("Recovery = ~p ms~n", [RT2]).
Recovery = 11 ms
ok
(b1@brahims-MacBook-Pro)50> ts_step3:addNode(ts1, 'a1@brahims-MacBook-Pro').
ok
(b1@brahims-MacBook-Pro)51> RT3 = ts_runner:recovery_ms(ts1,'a1@brahims-MacBook-Pro', 10),
                            io:format("Recovery = ~p ms~n", [RT3]).
Recovery = 11 ms
ok
```

Capitolo 4

Measurements

4.1 Metodo di misura

Le misure sono state raccolte con un modulo dedicato (`ts_runner`), basato su:

- **timer:tc/3** per il tempo di esecuzione in microsecondi (μs).
- Media su N=1000 iterazioni.
- **Locale**: operazioni su `ts1` (master).
- **Remoto**: da nodo A verso master B con riferimento `{ts1,'b1@...'}`, usando `rd/3` e `in/3` con timeout (per evitare blocchi).
- **Recovery**: tempo percepito lato API in millisecondi (ms) misurato su B: si spegne A con `rpc:halt` e si polla `nodes/1` ogni 10 ms finché A scompare.

4.2 Tempi medi (Locale vs Remoto)

Tabella 4.1: Tempi medi di `out/rd/in` (microsecondi, μs)

Scenario	**out**	**rd**	**in**
Locale (B)	0.383	1.127	1.166
Remoto (A→B)	1.286	70.338	47.665

4.3 Recovery time (ms)

Tabella 4.2: Recovery time (millisecondi, ms)

Run	**RT1**	**RT2**	**RT3**	**Media**
Valori (ms)	*11*	*11*	*11*	*11*

Capitolo 5

Conclusioni

Ho implementato un TS con API base, timeout lato client e visibilità multi-nodo via proxy, senza replica dei dati. Le scelte privilegiano semplicità e chiarezza: un solo master, matching lineare, polling lato client. Le misure quantificano le differenze tra locale e remoto e il tempo di ripristino quando un nodo cade.