ADCC Exam - Tuple Space

Nicholas Kania - #326850,
Matteo Leopizzi - #326110,
Giada Pierucci - #331320

December 30, 2024

Contents

1 Project description

The project aims to implement a ***Tuple Space*** (TS), a shared memory abstraction where various processes can interact through message passing.

Once the **creation** of the Tuple Space (*new(Name)*) occurs and at least one node is **added** (*addNode(TS, Node)*), it becomes possible to perform two types of **read** operations (*in/rd*) and one **write** operation, *out(TS, Tuple)*, which always succeeds.

The read operations *in(TS, Pattern)* and *rd(TS, Pattern)* are both **blocking**, however only the former is destructive, meaning that when a tuple is read, it must be removed from the TS. Given the blocking nature of the read operations, an additional version of them can be defined by setting a **timeout**. As a result, if no tuple matches the requested pattern, the operation will wait only until the specified time period expires.

In addition to the functions mentioned above, it is also possible to **remove** a previously added node from the TS if it is no longer needed (*removeNode(TS, Node)*) and to observe all the nodes currently present in the TS through the *nodes(TS)* function.

2 Implementation choices

2.1 Whitelist

We decided to implement an ***ETS*** table in order to ensure that only authorized nodes can perform operations related to the Tuple Space. Access to the ETS table is *'private'*, meaning that only the process responsible for its creation (the manager) can access and edit it. Additionally, a *'set'* type table was chosen, ensuring that the keys are unique. Furthermore, it is specified that:

- If the whitelist is empty, the first node can only add itself.
- If the whitelist is not empty, any node attempting to add another must be authorized.
- A node must be authorized if it wants to remove another one.

2.2 Pending Request Queue

We chose a list structure to store all read requests (*in/rd*) that don't match a specific pattern submitted related to any tuple in the Tuple Space. This approach was adopted to ensure that these requests are not discarded; their patterns are re-evaluated whenever a new tuple is inserted instead. The pending request is removed from this queue, when the corresponding pattern is found. Furthermore, if the read operation is destructive (*in*), the matching tuple is removed from the Tuple Space. In order to manage the queue, we also implemented these functions:

- The removal of all requests related to a specific node when that node is removed from the whitelist.
- The removal of a specific request.

2.3 Tuple Space

Regarding the management of the Tuple Space, a ***DETS*** table has been chosen to ensure data persistence and an automatic save is performed every 60 seconds. This approach allows for the restoration of the Tuple Space without data loss in case the manager fails. To boost performances, we flagged the DETS with *ram_file* flag enabling it to work like an ETS.

2.4 Supervisor

To address the issue of the manager failing, we implemented a supervisor process. This actor monitors and restores the manager in the event of receiving an *'EXIT'* message from it. Additionally, if it receives a *'stop'* message, it removes the link with the manager and deactivates itself.

2.5 GenServer

In addition to the traditional approach, an alternative version was implemented using the GenServer behaviour for the definition and management of the Tuple Space. This approach was adopted because GenServer provides a standardized and modular interface for managing the life cycle of processes, allowing for a comparison of test results with the traditional strategy. In this case, the handling of asynchronous (cast) and synchronous (call) messages is delegated to GenServer APIs, making communication between client and server less error-prone and the system easier to maintain.

2.6 TrapExit

The TrapExit has been set in the initialization phase of the manager process, to maintain control if the Tuple Space or an other link goes down. This allows the manager to not fail upon the failure of a connected node and to properly handle the *error* message.

3 Project Structure

3.1 Modules

3.1.1 Traditional

Tuple Space Supervisor - *tss* module It is responsible for supervising the Tuple Space Manager (*tsm*) and, in the event of its failure, the supervisor restores it.

Tuple Space Manager - *tsm* module Initialization functions:

- *init(Name, true)*: Create the Tuple Space and the supervisor process for the first time.
- *init(Name, Supervisor)*: Initialization of the manager, excluding the creation of the supervisor process. Used to restore the system upon failure.

It deals with the creation and management of:

1. an **ETS table**: whitelist;
2. a **DETS table**: store the Tuple Space on disk;
3. a **list**: the list of pending requests.

It manages messages sent through the Tuple Space Client functions. If the supervisor fails, the manager restores it.

Tuple Space Client - *ts* module Client interface where various operations are invoked and subsequently sent to the Tuple Space Manager.

Module *tstest* A suite of stress tests designed to evaluate the system's performance and resilience.

3.1.2 GenServer

The structure remains the same as the traditional approach, except for the **tsm** module, which is replaced by **tsb** and implements the gen-server behavior functions.

3.2 Data structure

- **Tuple Space (TS)**: Stores tuples inserted by authorized nodes. A DETS table is used for the periodic persistence of the Tuple Space on disk.
- **whitelist (WL)**: Stores the PIDs of authorized nodes, specifically those allowed to access the Tuple Space. This is implemented as a private ETS table, ensuring it is not exposed to external nodes. It is of type *'set'*, where the unique keys are the PIDs of the authorized nodes.
- **PendingRequestsQueue**: A list that contains read operations (*'in', 'rd'*) awaiting for a pattern to be matched.

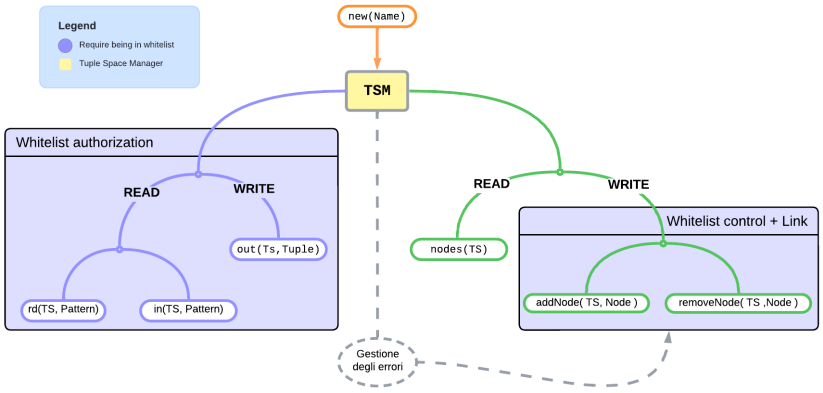

Figure 1: Flowchart illustration about mandatory functions in the Tuple Space Manager.

3.3 Functions

3.3.1 Client

- *new(Name)*: Creates a new Tuple Space with the given name.
- *in(TS, Pattern, Timeout)*: A read operation message performed on the Tuple Space, during which the corresponding element is removed from the table. A timeout value is provided to cancel the request in case of no response. If the specified pattern is not present in the Tuple Space, the read request is added to the Pending Requests Queue.
- *in(TS, Pattern)*: The same previous function, with *'Timeout = infinity'*.
- *rd(TS, Pattern, Timeout)*: Like the *in* function, but does not remove the tuple from the Tuple Space.
- *rd(TS, Pattern)*: The same previous function, with *'Timeout = infinity'*.
- *out(TS, Tuple)*: Writes the given Tuple on the Tuple Space indentified by TS. This function is always executed successfully.
- *addNode(TS, Node)*: Adds Node to the whitelist of the Tuple Space TS, getting authorization.
- *removeNode(TS, Node)*: Removes Node from the whitelist on the Tuple Space TS.
- *nodes(TS, Node)*: Returns the list of authorized nodes in the whitelist.

3.3.2 Server

- {*'EXIT', Supervisor, _Reason*}: Handles the supervisor crash.
- {*'EXIT', Pid, _Reason*} : Handles the client crash.
- {*in, Pid, Pattern*}: Handles the destructive read. Requires authentication.
- {*rd, Pid, Pattern*}: Handles the non-destructive read. Requires authentication.
- {*abort,* {*Type, Pid, Pattern*}}: Handles the abortion of a specific in/rd request. Requires authentication.
- {*out, Pid, Tuple*}: Handles the write operation to add the tuple to the TS. After this, try to execute requests in the Pending Request Queue. Requires authentication.
- {*add_node, Pid, Pid*}: Add the current node to the whitelist (if it's empty).
- {*add_node, Pid, Node*}: Handle the addition of the Node to the whitelist (standard). Requires authentication.
- {*rm_node, Pid, Node*}: Handle the removal of the Node from the whitelist. Requires authentication.
- {*nodes, Pid*}: Return the list of all authenticated nodes.
- {*stop, Pid*}: Handle the stopping and closing of the Tuple Space, notifying the supervisor for expected stop in order to not be respawned. Requires authentication.
- { _ } : Wildcard, ignores and removes from the queue all the unhandled messages.

3.3.3 Auxiliary

- *removeFromWhiteList(WhiteListRef, Pid)*: Removes the node from the whitelist, revoking its authorization to access the Tuple Space.
- *tryProcessRequest(TupleSpaceRef,* {*Type, Pid, Pattern*}*, PendingRequestsQueue)*: Checks whether an *in* or *rd* request for a specific pattern exists in the Tuple Space. If not, the new Pending Requests Queue is returned, containing the current request. If this request is a *in* one, we append it to the tail; otherwise it is appended to the head of the queue.

```
1 % Attempt to process a request based on the type (in/rd) and
      pattern
2 tryProcessRequest(TupleSpaceRef, {Type, Pid, Pattern},
      PendingRequestsQueue) ->
3     % Control over Pattern Matching
```

```
    Res = dets:match_object(TupleSpaceRef, {Pattern}),
    case Res of
        [] ->
            % 'in' requests are appended to the tail, while 'rd'
                ones are put as head of the queue, allowing
                grouping of pending requests
            case Type of
                in -> NewPendingRequestsQueue = PendingRequestsQueue
                    ++ [{Type, Pid, Pattern}];
                rd -> NewPendingRequestsQueue = [{Type, Pid,
                    Pattern}] ++ PendingRequestsQueue;
                _ -> NewPendingRequestsQueue = PendingRequestsQueue
            end;
        [{H} | _T] ->
            % Otherwise, process the request
            Pid!{ok, H},
            case Type of
                % if it's a destructive read, delete the tuple
                in -> dets:delete_object(TupleSpaceRef, {H});
                rd -> ok
            end,
            NewPendingRequestsQueue = PendingRequestsQueue
    end,
    NewPendingRequestsQueue
.
```

- *inWhiteList(WhiteListRef, Pid)*: Verifies the presence of a node into the whiteList, determining whether the node is authorized to access the Tuple Space.

```
% Check if the node is in the whitelist
inWhiteList(WhiteListRef, Node) ->
    Res = ets:match_object(WhiteListRef, {Node}),
    case Res of
        [] ->
            % If no match is found, set to false
            Present = false;
        [_H | _T] ->
            % If a match is found, set to true
            Present = true
    end,
    Present
.
```

- *removePendingRequests(PendingRequestsQueue, Pid)*: Removes all requests associated with the specified node from the Pending Requests Queue.

```
% Remove pending requests from the PendingRequestsQueue related to
    a specific Node
removePendingRequests(PendingRequestsQueue, Node) ->
    % Create a new list
    NewPendingRequestsQueue = lists:foldr(
        fun({_Type, Pid, _Pattern}, Acc) ->
            case Pid of
                Node ->
                    % If there's a match, do not include the related
                        request
                    Acc;
                _ ->
                    % Otherwise, include the request in the list
                    Acc ++ [{_Type, Pid, _Pattern}]
            end
        end,
        [],
        PendingRequestsQueue
    ),
    % Return the updated queue
    NewPendingRequestsQueue
.
```

- *abortPendingRequests({Type, Pid, Pattern}, PendingRequestsQueue)*: Removes from the Pending Requests Queue the submitted request.

```
% Remove a specific request from the PendingRequestsQueue
abortPendingRequest({Type, Pid, Pattern}, PendingRequestsQueue) ->
    NewPendingRequestsQueue = lists:foldr(
        fun(Elem, Acc) ->
                % Check if the current Elem matches the request to
                    abort
            case Elem of
                {Type, {Pid, _Tag}, Pattern} ->
                    % If it matches, it's not added to the new list
                    Acc;
                _ ->
                    % Otherwise, add it to the new list
                    Acc ++ [Elem]
            end
        end,
        [ ],
        PendingRequestsQueue
    ),
    NewPendingRequestsQueue
.
```

- *processPendingRequests(TupleSpaceRef, PendingRequestsQueue)*: Attempts

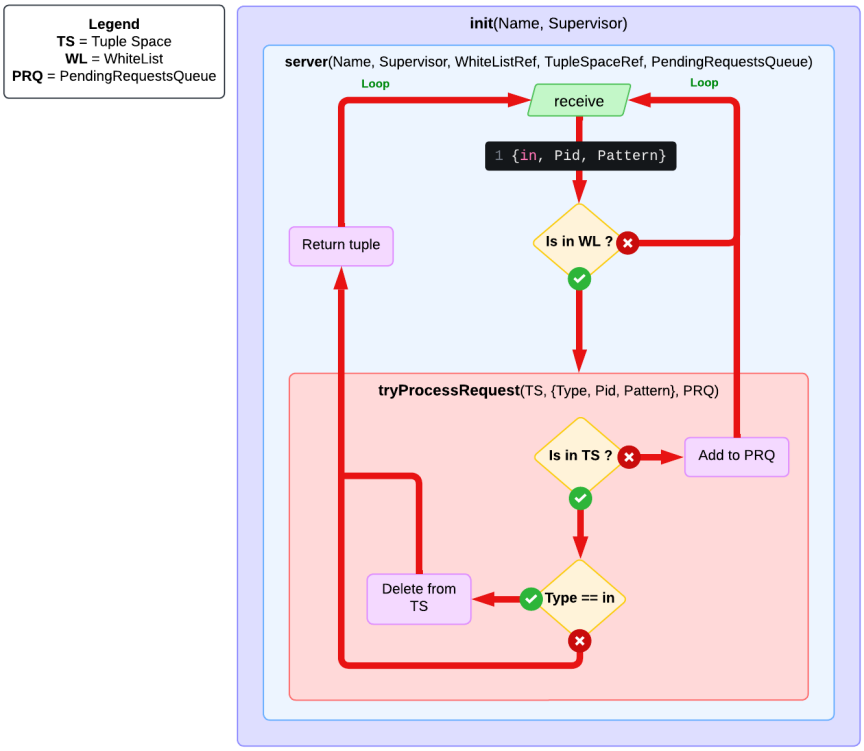

Figure 2: Flowchart about the *in* function.

to execute pending requests by invoking *tryProcessRequest*. Constructs a new Pending Requests Queue containing all requests that cannot yet be processed.

- *addNode(WhiteListRef, Node)*: Establishes a link between the Node (*ts*) and the Tuple Space Manager (*tsm*), and registers the Node into the whitelist.
- *removeNode(WhiteListRef, Node)*: Removes the link associated with the Tuple Space Manager (*tsm*), deletes the node from the whitelist, and clears all requests related to that node in the Pending Requests Queue.
- *getNodes(WhiteListRef)*: Retrieves the list of authorized nodes currently present in the whitelist.

4 Testing

In order to execute the testing functions, we assume that the node invoking the function to calculate the average read or write time is an authorized node. Therefore, the nodes created by these functions are added to the whitelist.

To carry out the necessary tests, three primary functions have been implemented. Two of these focus on calculating the average time required for executing read and write operations. The first function manages the sequential execution of the *out*, *rd*, and *in* operations, while the second handles *out* and *in* operations in a concurrent setting. In the concurrent mode, we ensure that all nodes are added to the whitelist, permitting only authorized actors to send requests to the Tuple Space. Lastly, the third function calculates the average time needed to restore the manager process upon scheduled failures.

4.1 Sequential

```
% Create a new tuple space
ts:new(myts).

% Run sequential test battery for I/O
tstest:testBattery_IO_seq(myts, 10000).
```

The sequential test was performed on a single actor, executing the *out*, *rd*, and *in* operations in order. This approach is justified by the fact that the *rd* function, since it's a non-destructive, it can be safely executed before the *in* operation on the Tuple Space.

Table 1: Average time for execute (sequential)

Version	# Ops.	Avg. OUT (μs)	Avg. RD (μs)	Avg. IN (μs)
Traditional	1	7.0	553.5	346.0
	10	2.3	257.4	191.7
	100	2.2	170.5	141.5
	1000	3.3	120.1	169.1
	10000	10.7	128.6	121.9
	100000	2.3	131.2	131.1
GenServer	1	5.5	346.5	330.5
	10	3.1	355.3	328.2
	100	4.2	251.6	267.0
	1000	12.7	194.9	162.0
	10000	12.5	135.0	153.9
	100000	4.0	161.7	172.4

4.2 Concurrency

```
% Create a new Tuple Space
ts:new(myts).

% Run concurrent test battery for I/O
tstest:testBattery_IO_conc(myts, 10000).
tstest:testBattery_IO_conc_2(myts, 10000).
```

With regard to concurrent (or distributed) testing, the operations performed are *out* and *in*, as concurrency would not ensure proper execution in the absence of a timeout for *rd* and *in* operations (Table 2).

Table 2: Average time for execution (concurrency)

Version	# Ops.	Avg. OUT (μs)	Avg. IN (μs)
Traditional	1	6.0	353.0
	10	5.2	214.5
	100	4.5	230.5
	1000	13.6	168.8
	10000	12.5	146.0
	100000	3.0	138.4
GenServer	1	5.0	661.5
	10	3.1	270.5
	100	4.2	218.4
	1000	11.9	210.6
	10000	4.0	174.3
	100000	2.1	184.5

Table 3: Average time for execution (concurrency)

Version	# Ops.	Avg. OUT (μs)	Avg. IN
Traditional	1	53.5	0.2 ms
	10	130.1	0.5 ms
	100	223.0	63.7 ms
	1000	893.9	2.8 s
	10000	4028.9	24.0 s
	100000	108551.0	118 s

In this version, each operation is performed by a different node, and the system manages all the connections between the nodes simultaneously (Table 3).

4.3 Failure Recovery

We chose to test the failure recovery time of the Tuple Space Manager, which is restored by the supervisor when it fails.

```
% Create a new Tuple Space
ts:new(myts).

% Run test for estimating average recovery time after Tuple Space
    Manager fails
tstest:avgTimeRecovery(myts, 10000).
```

Table 4: Average time of recovery

Version	# Ops.	Avg. Recovery (μs)
Traditional	1	15150.0
	10	6677.8
	100	4931.6
	1000	4657.6
	10000	4713.6
	100000	5446.3
GenServer	1	24415.5
	10	10256.2
	100	7397.8
	1000	6904.9
	10000	6866.7
	100000	7679.8