Efficient Tuple Space Mobile Agent-based Reservation System

Vikas Aggarwal
Balaji Krishnamachari-Sampath
Deepak Rao

Abstract

The project involves Efficient Tuple Space implementation, in a distributed environment, on top of agent architecture. Agent architecture is implemented and is used to send agents to different machines to fetch the tuple, if the tuple is not present in the present Tuple Space Manager. The agent migrates to the different hosts depending upon policies. The number of tuples stored in a Tuple Space Manager and the number of queries satisfied by a host for a tuple type are used as the metrics to find the next host (to which the agent has to migrate in order to fetch the tuple).
**Functionality of a single Tuple Space Manager**

- **Application**
  - `In()`
  - `out()`
  - `In()` is blocking RMI call

- **TSM**
  - `In()` (Non-Blocking)
  - `Out()`

- **Hash Table**
Overview of the complete functionality of our system

Unblock the In() call

Send Query Agent

Invoke Query Tuple

Retun Result Agent, match is found

Agent manager 0

Agent Manager 1

Block the In() call
In ()

Block the In() call

Send Query Tuple to the next node (obtained from current policy)

Invoke Query Tuple () of the TSM of next node

Match found

Have all nodes in the cluster been visited?

Send a Return agent with the matching tuple back to the node that requested the tuple

Unblock the In call

End
Interaction of Tuple Space Manager with the Hash table of tuples and the Vector table of nodes

Tuple Space Manager

Hash Table
- Object Type1
- Object Type2
- Object Type3

Vector Table

<table>
<thead>
<tr>
<th>Host Name</th>
<th>Object Type</th>
<th>Num of Objects</th>
<th>No of Query Satisfied</th>
</tr>
</thead>
</table>
The project deals with the problem of developing Efficient Mobile Agent migration policies that take into account computing resources. Such resources may include, not only computing resources available at the node, but also the associated cost to reach a given node over the network. Migrating an agent to a node that doesn’t have a necessary resource is costly, both in terms of computation as well as time. There is also a high network associated with dispatching an agent to that node. The agents can be migrated efficiently by knowing the cost of reaching different nodes and also number of resources available at each and every node. The project attempts to study the development of such agent migration policies, with the context of implementing a tuple space.

**Overview of the System:**

The tuple space will be implemented as a collection of Tuple Space Managers (TSM) residing on each machine. Mobile agents will be used to retrieve the data tuple from different TSMs. Agent architecture will provide a framework to create and migrate agents to other machines. Further, the Agent Architecture resides on an adaptive middleware capable of providing notifications on resource levels at various clients and also a policy manager to change the policies, that is, the agent migration policies.

<table>
<thead>
<tr>
<th>Tuple Space</th>
<th>Adaptive Middleware</th>
<th>Agent Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Software stack of the proposed Tuple Space*

The tuple space is actually made of multiple tuple space managers (TSM), each residing in its own machine. For our project, computing resources are represented as tuples of different types, stored by each TSM. In each TSM, there is a hash table, which stores the tuples in its own tuple space manager. Also a vector table is stored at each TSM, which stores the number of objects of a particular type in other TSM of its cluster, and the number of queries satisfies by the other TSM's. The vector table does not store the information of all TSM about all the objects. Only information about those objects that the TSM requires is
recorded. Counts are maintained using a middleware layer to notify other TSM’s when tuples of interest fall below a certain level/ goes above a certain level. Also if a node has a certain tuple does not mean that there is a match. The node may be having a tuple of object type 1 but it may not be having the corresponding tuple. The notifications are sent to the interested tuple space for only change of object type not for the individual tuple.

**Agent Architecture and Tuple Space Manager (TSM)**

Agents are Java objects, in order to enable them to execute in a separate thread, agents are required to implement the `Runnable` interface.

TSM on each host stores two tables. First is a Hash table storing the tuples of the object types and the second is a vector table storing the number of objects of each type on other hosts (as per the latest information on this host).

When an application requests for matching a particular tuple, TSM first checks it in its Tuple Space. If match is not found, then it finds the next node information from the Vector table depending upon the policy (Policy can be to send the agent to the node having the largest number of that object type, or recent node from which objects of that particular type were obtained). It then sends this information to the Middleware via a Remote Method Invocation (RMI) call where the Agent Architecture (AA) creates an agent and sends it to the next node.

When a node receives an agent, the AA launches the agent in a separate thread. The agent then makes a RMI call to the TSM (on the migrated node) requesting information about the tuple. Incase a match is found; another agent is sent to the source node, through the AA. Otherwise, the agent is migrated to the next node based on the policy in the present TSM, leaving behind the information in TSM indicating that an agent has visited the node and it was not able to find the matching tuple for a tuple T (tuple T is stored along with the source node identifier).

When the TSM finds match for a query tuple, the particular tuple is removed from the Hashtable of that TSM, update the table that contains the number of the objects that it has to reflect the change. When two nodes give tuples for the same query tuple (that is satisfy the same request), the node that originated the request will receive more than one tuple. It should then update its table to reflect the change.

Since we are leaving a trail of the agent in nodes that do not contain a matching tuple, a possibility exists that the node does not contain the tuple when the agent arrives, but it gets the tuple after the agent has moved to another node. In the meantime, the agent could have gone to some other node and that node could be having the tuple and hence both of the nodes would be sending the tuple to the source node (that is the originator). In this case, the originator of the tuple will be getting more than one tuple, it uses one of them to satisfy the query and make all of the other tuples as its out-tuple (that is to store them in its tuple space).
space), it also modifies the table to reflect the changes in number of tuples.

Types of policies

The policies in a TSM are mainly those concerning agent migration; that is where to start looking for a particular tuple? We feel the availability of resource (in our case the tuple) will dictate the policy. In this respect, we have identified two types of policies:

1. **Object count policy:** Agent migrates to the node having the maximum number of tuples of a particular type T.

2. **Queries satisfied policy:** Agent migrates to a node that has satisfied maximum number of client requests for type T.

The performance of these policies should be better than an agent migrating to machines in a fixed order. The number of hops the agent takes to satisfy a request is a fair indication of how a policy can affect the performance.

Application perspective

This section discusses the performance of application (TSM) with and without adaptation policies. The performance of TSM is measured for each different type of policy.

In a TSM, choosing the next node for agent to hop to can be considered as a policy (agent migration policy). The default policy of the application in a node is to migrate the agent to a fixed next node. On simulating the TSM application for 3 replications of 500 ‘in’ operations produced the graphs in Appendix 1. Number of hop counts by an agent to find the tuple of interest is a fair indication to measure the effectiveness of a policy.

TSM is simulated for three types of policy transitions (or policy changes):

a) The agent always migrates to a fixed next node using default policy.

b) Agent migration policy to fetch a particular tuple (T) changes from default policy to an object count policy when the number of tuples of type T, in a remote TSM, exceeds 10. Note that there is no transition of policy back to the default policy.

c) Agent migration policy to fetch a particular tuple (T) changes from default policy to an object count policy when the number of tuples of type T, in a remote TSM, exceeds 10. Or the policy changes to ‘queries satisfied policy’ if the number of queries satisfied by a TSM
exceed 10. Note that there is no transition of policy back to the default policy.

Performance of application without policy management

As can be observed from the graphs in Appendix 1, hop count necessitated while following the default policy is greater than that necessitated by other policies. This is the case with all three graphs denoting Distributed-in Single-out, Single-in Distributed-out and Distributed-in Distributed-out simulations. Hence from this observation we can infer that performance of an application depends on the type of policy chosen.

Performance of application for each type of policy

From the graph for Distributed-in Single-out, we observe that lesser number of hops is needed for an agent while following the combination of queries satisfied policy and object count policy. While the number of hops needed for following default policy alone is the greatest.

Similarly, from the graph for Single-in Distributed-out simulation, we find that combination of queries satisfied policy and object count policy yields the best result (that is the least hop count). Another observation is the close correspondence in the lines for default policy and object count policies. Consider the scenario where application on node A is requesting (in operation) tuples of type T. On not finding a tuple of type T in its own TSM, A could dispatch agents
to all the nodes in the tuple space. Thus creating a trial of requests for tuple of type T. Any node on receiving a tuple of type T, immediately dispatches it to A. Due to the trial of requests, all the nodes on receiving tuples of type T, send them to A. Thus reducing the scope for object count of any node, for tuple of type T, to increase beyond 10. Thus resulting in usage of primarily the default policy.

The two simulations considered above are the extreme cases that rarely exist in reality. Simulations for the more prevalent case of Distributed-in Distributed-out, showed the best result. From the graph for Distributed-in Distributed-out simulation in Appendix 1 we can observe a marked improvement in using object count policy, the number of hops reduced by 50.5% when compared with that of default policy. Further improvement can be observed for the combination of queries satisfied policy and object count policy. The latter case shows hop count reduction of around 72% than the default policy. Similarly, hop count reduction of 44.5% is observed when compared to object count policy alone.

Number of replications observed was insufficient to draw conclusions based on statistical analysis. However, in general the combination of object count policy and queries satisfied policy fared better than default policy and object count policy individually. Similarly, in general, object count policy fared better than the default policy, with the worst case being a marginal improvement over that of default policy (as in the case of Single-in Distributed-out simulation).

Lesson learnt
1. Functioning of a distributed Tuple Space Manager
2. Concept of mobile agents
3. Exposure to JAVA environment.
4. Working of JAVA RMI calls
5. Functionality of thread, synchronization mechanisms used by JAVA

Enhancement

Implemented the Tuple Space Manager in a single cluster of nodes. In case a match is not found in the cluster, then an agent needs to be sent to the next cluster head and a similar search has to be carried out in the next cluster. We have not implemented this functionality in our project.

Also the policies for routing the agents have to depend on the network load and bandwidth supplied by different mediums. We have not considered the network properties for the design of our application.
Conclusions

Important conclusions can be drawn based on the simulation results obtained above.

- The middleware provides better performance for applications like mobile agents.
- Precise amount of performance gain is dependent on the type of application and also on the types of policies being used, the performance gain varied from 4 to 72% depending on the type of policy.