Tuple Space in the Cloud

Hariprasad Hari

Institutionen för informationsteknologi
Department of Information Technology
Abstract

Tuple Space in the Cloud

Hariprasad Hari

Communication Overhead is one of the major problems hindering the prospects of emerging technologies. There are various mechanisms used for communication and coordination of data in the form of Message Passing Interface and Remote method Invocation but they also possess some drawbacks. Tuple Space is one such candidate mechanism used in parallel processing and data sharing. So far there were been many centralized Tuple Space implementations such as Java Space, TSpace. As the communication between processes increases, the centralized communication becomes a bottleneck and hence there is need to distribute the processes. A better way is to distribute the Tuple Space itself. The goal of this thesis is used to find out the problems which would arise when distributing the Tuple Space. It analysis the state-of-art tuple space implementations and provides a novel approach by proposing a solution which satisfies most of the problems.
 Tuple Space in the Cloud

PER KARLSSON MScEE
Manager CDN & Cloud Computing Research

JOACIM HALÉN
Senior Researcher

Ericsson AB, Packet Technologies Ericsson Research, Färögatan 6, 164 80, Kista, Sweden.
Acknowledgements

I have few people to thank who have made this thesis possible and complete it successfully on time. I would like to thank starting from Olle Gällmo, Lecturer, Uppsala University who initially gave me an opportunity to take part at Erlang User Conference, Stockholm at October 2010. It is due to this conference I came to know about more people from the industry arena particularly from Ericsson AB.

Next, I wish to thank Johan Kolhi, Researcher, Ericsson Research, Kista for giving me moral support and a start for my thesis by introducing me to his manager, who is Per Karlsson P, Manager, Ericsson Research, Packet Technologies and CDN, Kista. Without that meeting between me and Johan during the conference, this could not have been achieved.

I whole heartedly thank Pelle (Per) who gave me this opportunity to work as a Researcher in a field which I was interested. I thank Jochim Halén, Senior Researcher, Ericsson Research, Kista, my supervisor, who guided, motivated me throughout my thesis period. He and Pelle would always give me ideas, suggestions regarding my thesis work. I should also thank them for calling me, to all the section meetings, fikas, lunch which were taking place at the department. Those things made me to work better. It is good to work at Ericsson, as I learnt new things everyday and it paid a way to keep myself updated in today’s emerging technologies.

I extend my thanking list to all the members of the department especially Hareesh, Enrique for their timely help and support during my thesis. Finally I would like to thank the Ericsson Research department for providing me essential resources for carrying out my thesis work.

I would like to say a special thanks to my reviewer Arnold Pears, Senior Lecturer, Uppsala University for reviewing my work.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Problem Definition ............................................. 1  
1.2 Research Approach ............................................. 2  
1.3 Thesis Outline ................................................ 2

## 2 Fundamental Concepts

2.1 Origin of Tuple Space .......................................... 3  
2.2 Tuple Space .................................................... 4  
2.3 Tuple .......................................................... 5  
2.4 Template ........................................................ 6  
2.5 Primitives of Tuple Space ..................................... 7  
  2.5.1 OUT ............................................................ 7  
  2.5.2 IN ............................................................. 8  
  2.5.3 RD ............................................................ 9  
  2.5.4 EVAL .......................................................... 10  
2.6 Multiple Tuple Space Primitives ............................. 10  
  2.6.1 Copy_Collect ................................................. 10  
  2.6.2 Collect ..................................................... 10  
2.7 An Application - Mergesort ................................... 10  
  2.7.1 Description ................................................ 11  
2.8 Summary ....................................................... 12

## 3 Analysis of Candidates

3.1 Problem Statement .............................................. 13  
3.2 Eligibility Criteria ............................................ 15  
3.3 York Kernel ..................................................... 15  
  3.3.1 Short conclusion ......................................... 16  
3.4 GSpace ........................................................ 17  
  3.4.1 Short Conclusion ......................................... 18  
3.5 Grinda .......................................................... 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future Work</td>
<td>51</td>
</tr>
<tr>
<td>Conclusion</td>
<td>53</td>
</tr>
<tr>
<td>Bibliography</td>
<td>55</td>
</tr>
<tr>
<td>A Merge Sort Application</td>
<td>59</td>
</tr>
<tr>
<td>A.1 Installation Manual</td>
<td>59</td>
</tr>
<tr>
<td>A.2 Source Code</td>
<td>60</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Shared and Distributed Memory Architectures</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>A Tuple Space Coordination Model</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>OUT operation</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>IN operation</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>RD operation</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Our Distributed Environment context</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Analysis Report</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Current Tupleware Architecture[14]</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Tupleware Architecture extended</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Specific geographic location</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Super Peer Architecture</td>
<td>39</td>
</tr>
<tr>
<td>5.3</td>
<td>Peer Node</td>
<td>40</td>
</tr>
<tr>
<td>5.4</td>
<td>Super Peer Node</td>
<td>41</td>
</tr>
<tr>
<td>5.5</td>
<td>Tuple Management Component</td>
<td>42</td>
</tr>
<tr>
<td>5.6</td>
<td>Storage of data inside Mnesia and Relational database</td>
<td>43</td>
</tr>
<tr>
<td>5.7</td>
<td>Control Component</td>
<td>44</td>
</tr>
<tr>
<td>5.8</td>
<td>Super Peer layout</td>
<td>45</td>
</tr>
<tr>
<td>5.9</td>
<td>Supervision Tree Level</td>
<td>46</td>
</tr>
<tr>
<td>5.10</td>
<td>Network Management Component</td>
<td>46</td>
</tr>
<tr>
<td>5.11</td>
<td>Simple Communication Flow</td>
<td>47</td>
</tr>
<tr>
<td>5.12</td>
<td>Client’s View to the System</td>
<td>48</td>
</tr>
<tr>
<td>5.13</td>
<td>Tuple Storage</td>
<td>48</td>
</tr>
<tr>
<td>A.1</td>
<td>logo</td>
<td>69</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Difference between Tuple space model and MPI model</td>
<td>5</td>
</tr>
<tr>
<td>4.1</td>
<td>Satisfaction Table</td>
<td>32</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The Computer Industry seems to keep on changing the underlying Technology it uses. It ranges from traditional Personal computing, Client-Server model, Parallel computing, Distributed computing to the recent Utility computing and Cloud computing. These are all the “buzz” words in today’s software market which can do wonders and change the world around us.

The major growth in Information Communication Technologies (ICTs) for the past two decades or more has resulted an exponential increase[1] in the computing power, communications usage and other resources such has hardware, maintenance of data centers, managing large number of clients. To avoid the effects caused by the increase in computing power and to control the resources cost many IT majors and small companies have moved their applications and infrastructure to the Cloud or considered to be moved to the Cloud. The main advantage of the Cloud is that you can use the resource such as hardware, networks, storage, software, business service when you need them and pay for the resource which you use. Companies like Amazon, Eucalyptus, Ericsson, Google, Microsoft, IBM, etc and open source projects such as OpenNebula, Openstack, etc have deployed their applications into the Cloud. As the Cloud has the advantage of providing Infrastructure, Platform or Application as a Service, the mode of deployment for many applications by these companies, ranges from distributed to parallel deployments. The Cloud can grow or shrink depending upon the requirements of the applications. Therefore these applications deployed needed to be executed in parallel and also across many geographical locations.

1.1 Problem Definition

The current market trend is that all the applications such as scientific applications, business applications, telecommunication applications, etc., are deployed either in a parallel environment or in a distributed environment or both. Deploying the applications in these environments poses some serious challenges to the software developers and maintainers who are making them to run smoothly and meet the customer needs. One of the major challenges would be to mitigate the Communication Overhead which arises due to the millions of communications over the Internet. Many mechanisms such as Message Passing Interface, Remote Method Invocation, Remote Procedure Call have been developed and provided to reduce the communication overhead of the data in the network and coordinate their actions smoothly. But these mechanisms also seems to have some limitations.

So this thesis work is aimed to study one such mechanism for coordination of parallel processing and data sharing known as Tuple Space proposed by David Gelernter in the LINDA programming language. It is a coordination model which aims to solve the com-
communication overhead problem and it is well supported by its centralized implementations and parallel processing. But there are less theoretical or empirical evidence to showcase that Tuple Space model is also suitable in our context of distributed environment shown in fig 3.1.

The major aim of the thesis work is to study and analyze how the Tuple Space model can be used to solve the problems which arises due to distributed environment (i.e. one such problem is the communication overhead, more in section 3.1) so that any application can be deployed in a distributed environment using this model.

The main Objectives of the thesis are as follows:

- Identify the major problems when distributing Tuple Space.
- Analyse and evaluate all the possible solutions for these problems.
- Provide a novel design in extending the most possible candidates or a new approach which can be used to solve those problems.

1.2 Research Approach

The thesis work have been carried out in several stages throughout the entire period. The majority portion and the main part of my thesis work is research and analysis. This thesis work takes more of a theoretical study about Tuple Space.

The first stage is studying about the tuple space itself.

The second stage is studying many research papers, articles, whitepapers about the various implementations of tuple space to know about the problems they have considered and the solutions given to those problems.

The third stage is providing the analysis of the various candidates based on the research conducted in the second stage and give a analytical report.

The fourth stage is to suggest a novel solution based on this research and analysis by extending the most suitable candidate solution or give a new idea.

1.3 Thesis Outline

The thesis is structured and organised into following chapters:

Chapter 1 Gives the introduction of the thesis.

Chapter 2 Describes the fundamental concepts such as Tuples, Templates and Tuple Space.

Chapter 3 Gives a detailed analysis of all the tuple space implementations that have been used to solve the problems. It also explains about the pros and cons of each of these techniques.

Chapter 4 Theoretically evaluates the candidates based on criteria’s.

Chapter 5 Describes the design and architecture of the new proposed solution.

Chapter 6 States the intended future work related to the thesis.

Chapter 7 Gives the conclusion of the thesis.
Chapter 2

Fundamental Concepts

This chapter provides background or fundamental knowledge that is essential for a reader to understand about the thesis. Section 2.1 briefly introduces the Origin of Tuple Space. Section 2.2 discusses about tuple space and its properties. Section 2.3 explains about Tuples. Templates are explained in section 2.4 followed by the primitives used in tuple space which are explained with suitable examples in section 2.5. A brief note about multiple tuple space primitives is given in section 2.6. A practical application to understand tuple space is provided in section 2.7.

2.1 Origin of Tuple Space

Computations and Communications are increasing exponentially day by day. Both need to be efficiently handled by the programmers to develop various applications. The size of the problem for various applications keeps on growing. Solving these large problems with the help of a single machine is not a feasible solution. We need to have more computational resources in order to solve these large problems simultaneously. One solution is breaking those larger problems into smaller ones and solving them concurrently. This is known as Parallel Computing. They play a major role in today’s communication systems and other scientific applications. There are many parallel programming models such as Shared Memory Architecture (SMA)\(^1\) 2.1, Distributed Memory model\(^2\) 2.1, Thread model and Hybrid model.

In each of these models, each process communicates with other processes in the system by having a common memory or bus topology. So the inter-process communication or the coordination between processes is done in many ways such as Message passing Interface (MPI), Remote Method Invocation (RMI), CORBA\(^3\) and Remote Procedure Call (RPC), etc. One such coordination model between processes is given by David Gelernter and Nicholas Carriero of York University in the year 1985. This coordination model is called as Tuple Space. This was first used in the programming language LINDA which is used for parallel computation/processing between objects. Linda is an abstract parallel programming language and it is independent of any machine architecture.

---

\(^1\)There is a central memory which is shared between one or two processes.

\(^2\)Each process has its own memory. This memory is shared between different processes through the network topology.

\(^3\)This model is used for developing distributed systems

---

3
2.2 Tuple Space

Tuple Space is a repository which is used to store/retrieve objects (i.e. data). It is an implementation of the Associative memory paradigm. The objects can be accessed through pattern matching of their contents due to the associative nature of the tuple space.

It is like a Persistent storage memory i.e. the objects can be stored in it until they are retrieved by some other processes. The process which stores the objects is called Producer. Example: process A and B in Figure 2.2. The process which retrieves the objects is called Consumer. Example: process A, B and C in Figure 2.2. There is no restriction that a process should be a producer or a consumer. Depending upon the action (storing/retrieving), they are named as producer or consumer. The inter-process communication i.e. between the producer and consumer is decoupled in Space and Time.

It is decoupled in space because any process can store the object which can be retrieved by any other process. This is an advantage from the conventional MPI model in which the sender must know the destination of the receiver. This is an important difference between tuple space and MPI model. For more differences refer table 2.1. Similarly it is decoupled in time because the receiver can retrieve the objects whenever he wants. The object does not belong to any process when it is inside the tuple space. Since it is decoupled in space and time it is known as Generative communication.

It is Logically shared memory because the objects are shared between different processes but there is no physical memory shared. Tuple space has an advantage where the communication and computation between the processes are separated. This is extremely useful for many parallel and mobile applications. It is an example of the master/worker pattern.

---

4 Associative memory is that the elements are accessed by their contents rather than by their address.
5 The data (objects) which is present inside the tuple space does not belong to any process.
6 It also works in ad hoc, peer-to-peer patterns.
pattern. Any process can be a master. The master process sends many tasks to the tuple space which has to be carried out concurrently by the worker processes. Each worker process gets a task from the tuple space, executes it and gives the result of the task back to the tuple space. None of the worker process is dependent on other workers, so the task which is carried out in parallel is faster and efficient. Finally the master process collects all the results and gives back to the application.

2.3 Tuple

Tuples are the fundamental objects of a Tuple Space. It is a collection of ordered list of elements. Each element in a tuple is composed of field and value pairs. A Tuple can have any number of elements (i.e. any number of field and value pairs). The format of a Tuple is \((\text{field1, value1, field2, value2, field3, value3, } \ldots)\). NULL values are NOT allowed in a tuple. There are two types of tuples; Active and Passive tuples. All the tuples stored inside the tuple space are called as passive tuples. Active tuples are those

<table>
<thead>
<tr>
<th>No</th>
<th>Tuple Space model</th>
<th>Message Passing Interface model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Processes need not to be available at the time of communication.</td>
<td>Process has to be available during communication.</td>
</tr>
<tr>
<td>2</td>
<td>Process can exist without knowing other processes.</td>
<td>Has to know the receiver.</td>
</tr>
<tr>
<td>3</td>
<td>Any one can access the data.</td>
<td>Only the receiver can access the data.</td>
</tr>
<tr>
<td>4</td>
<td>It does not have ordering.</td>
<td>It has ordering of messages.</td>
</tr>
<tr>
<td>5</td>
<td>It is not a broadcast message.</td>
<td>It can broadcast messages.</td>
</tr>
<tr>
<td>6</td>
<td>It has the temporal advantage.</td>
<td>Messages once processed are lost.</td>
</tr>
</tbody>
</table>
which can spawn other process or perform some function along with the calling process concurrently inside the tuple space. Once they have finished, they turn into passive tuples. Passive tuples are stored in the tuple space until they are retrieved by some process. Examples of tuples are:

- \( t_1 \) \( (4, \text{linda}, 10.65) \),
- \( t_2 \) \( (\text{parallel}, \text{cloud}) \),
- \( t_3 \) \( (25, 100, 99) \) etc.

In the above examples the value is given along with the field (i.e. data type of the value is given in subscript). All the three tuples are of different patterns. For instance the tuple \( t_1 \) has a pattern \( (\text{linda}, \text{string}, \text{float}) \) with 3 fields, \( t_2 \) has a pattern \( (\text{parallel}, \text{string}) \) with 2 fields, \( t_3 \) has a pattern \( (\text{int}, \text{int}) \) with 3 fields and likewise we can have many tuples with different patterns and with many number of fields.

### 2.4 Template

**Templates** are tuples used for retrieval of tuples from the tuple space. They are also composed of field and value pairs. The format of a template is \( (\text{field1, value1, field2, NULL, field3, \ldots}) \). NULL values are allowed in a template. Examples of templates are:

- \( \text{temp1} \) \( (10, \text{NULL}) \),
- \( \text{temp2} \) \( (\text{book}, \text{NULL}, \text{NULL}) \),
- \( \text{temp3} \) \( (\text{NULL}, 25, 30, 300, \text{NULL}) \),
- \( \text{temp4} \) \( (\text{NULL, NULL, lan}, \text{parallel}) \),
- \( \text{temp5} \) \( (30.6) \) etc.

The templates can have actual and formal values. Formal values are those which will be replaced during matching process. Actual values are those which will be compared during matching process. Hence the place holder NULL is unique to templates as this would be replaced by that actual value when the template is matched against the tuple.

For instance, if we want to retrieve the tuple “t1” in section 2.3, we should give the template as \( \text{tempA} \) \( (4, \text{NULL}, 10.65) \). This will match the tuple “t1” and would retrieve the tuple and instead of the NULL placeholder it will have the value: \( \text{linda} \).

Likewise the templates for the other tuples would be:

- \( t_2 \) \( (\text{parallel}, \text{cloud}) \) \( \rightarrow \) \( \text{tempB} \) \( (\text{NULL, NULL}) \),
- \( t_3 \) \( (25, 100, 99) \) \( \rightarrow \) \( \text{tempC} \) \( (25, \text{NULL, NULL}) \),

To match the tuple we should have the exact pattern and the exact number of fields. Example:

- \( t_3 \) \( (25, 100, 99) \) \( \rightarrow \) \( \text{tempC} (25, \text{NULL, NULL}) \) [Correctly Matched]
- \( t_3 \) \( (25, 100, 99) \) \( \rightarrow \) \( \text{tempC} (25, \text{NULL, NULL}) \) [Incorrect number of fields]
- \( t_2 \) \( (\text{parallel}, \text{cloud}) \) \( \rightarrow \) \( \text{tempC} (\text{lan, NULL}) \) [Incorrect pattern]
2.5 Primitives of Tuple Space

To store and retrieve tuples from tuple space, LINDA provides four basic primitive operations: OUT(2.5.1), IN(2.5.2), RD(2.5.3) and EVAL(2.5.4). These are as follows:

2.5.1 OUT

The “out” primitive is used by any process who wants to store a tuple in the tuple space. The format of the “out” primitive is \(<out\ (field1, value1, field2, value2, \ldots )>\). The argument passed to the “out” primitive is a tuple. It does not return anything i.e. any process which uses an “out” primitive will surely store the tuple in the tuple space. This can be seen from diagram 2.3 where three processes A, B and C performs “out” operation.

![Diagram showing OUT operation with processes A, B, and C storing tuples in the tuple space.]

There is an ensured ordering between two “out” in a process, i.e. when there are two sequential “out” operations by a process; it is ensured that the first “out” operation would be executed and store the tuple in the tuple space before the second “out” operation starts. A process can store only one tuple by using this primitive at a time. If a process wants to store many tuples then it needs to call the “out” primitive many times. i.e. the “out” primitive cannot be used like this \(<out\ (100_{\text{int}}, 9_{\text{int}}), (10_{\text{int}}, 99_{\text{int}})>)\.

Examples for “out” primitives are: Let’s say there are six tuples which are going to be stored in the tuple space.

- \(<out\ (100_{\text{int}}, 9_{\text{int}})>)\,
- \(<out\ (10_{\text{int}}, 99_{\text{int}})>)\,
- \(<out\ (4_{\text{int}}, \text{linda}_{\text{string}}, 10.65_{\text{float}})>)\,
- \(<out\ (\text{parallel}_{\text{string}}, \text{cloud}_{\text{string}})>)\,\,
- \(<out\ (25, 100, 99)>)\,
- \(<out\ (90)>)\,
- \(<out\ (25, 100, 99)>)\,\,

7
2.5. Primitives of Tuple Space

• \(<\text{out} (25_{\text{int}}, 100_{\text{int}}, 99_{\text{int}})>,\)
• \(<\text{out} (90_{\text{int}})\>) etc.

2.5.2 \textbf{IN}

The “in” primitive is used by a process who wants to retrieve a tuple from tuple space. It is used to permanently remove the tuple from the tuple space. The format of the “in” primitive is \(<\text{in} (\text{field1}, \text{value1}, \text{field2}, \ldots)\>). The argument passed to the “in” primitive is a template. This template should match with the tuples that are already stored in the tuple space. Normally the first tuple which is matched with the template is retrieved and returned back to the process which has called the “in” operation. Based on the return value there are two forms of this primitive. First is the “blocking in” primitive and second is the “non-blocking in” primitive. A process cannot remove more than one tuple from the tuple space at a time. If it needs to remove more than one then it has to give successive “in” operations. The diagram 2.4 shows an “in” operation performed by three processes A, B and C.

![IN operation diagram](note: This is a text representation and cannot display the diagram. The text describes the diagram.

If there is no matching tuple found in the tuple space then the process which has called this operation waits until it gets the matching tuple. This form of “in” operation is known as “blocking in” operation. This operation is synchronous. The non-blocking primitive returns NULL back to the process which has called the “inp” operation. This operation is asynchronous.

Examples for “in” and “inp” primitives are:

1. \(<\text{in} (\text{100}_{\text{int}}, \text{NULL})> \rightarrow (\text{100}_{\text{int}}, 9_{\text{int}}),\>
2. \(<\text{in} (\text{4}_{\text{int}}, \text{NULL}, \text{NULL})> \rightarrow (\text{4}_{\text{int}}, \text{linda}_{\text{string}}, 10.65_{\text{float}}) \text{ or } (\text{4}_{\text{int}}, \text{100}_{\text{int}}, 99_{\text{int}}),\>
3. \(<\text{in} (\text{30.5}_{\text{float}})> \rightarrow \text{waiting},\>
4. \(<\text{inp} \ (30.5_{\text{float}})> \rightarrow \text{NULL etc.}\)

In example (2), even though the template matches both the tuples, only the first tuple which is matched would be returned back to the process i.e. the tuples are chosen non-deterministically (process B). If we want to retrieve the other matched tuple the process has to call the “in” operation again. In example (3), there is no tuple matching the template. Hence the process keeps on waiting (process C) and in example (4), the “inp” primitive returns NULL back to the calling process (process A). From the figure, process C keeps on waiting as it queried for the tuple \(<\text{in}(30.5)>\) which is not present in the tuple space.

### 2.5.3 RD

The “rd” primitive is used by a process who wants to read a tuple from tuple space. The format of the “rd” primitive is \(<\text{rd} \ (\text{field1, value1, field2, NULL, ...})>\). The argument passed to the “rd” primitive is a template. This template should match with the tuples that are already stored in the tuple space. This is also similar to the “in” primitive but it does not remove the tuple from the tuple space rather gets a copy of the tuple and returns back to the calling process. It also has two forms of primitive based on the return value. The blocking read “rd” and the non-blocking read “rdp” primitives. These are similar to “in” and “inp” except they don’t remove the tuples. The diagram 2.5 shows an “rd” operation performed by three processes A, B and C.

![RD operation diagram](image-url)

Examples for “rd” and “rdp” primitives are:

- \(<\text{rd} \ (100, \text{NULL})> \rightarrow \ (100, 9)\),
- \(<\text{rd} \ (4, \text{NULL, NULL})> \rightarrow \ (4, \text{linda}, 10.65) \text{ or } (4, 100, 99)\),
- \(<\text{rd} \ (30.5)> \rightarrow \text{waiting}\),
2.6 Multiple Tuple Space Primitives

Linda programming language is an abstraction and is architecture independent. The above said basic primitives can be used only when there is a single tuple space. This is a limitation in the basic Linda model when two or more processes has a template which matches with more than one tuple in the tuple space. Then using multiple “rd” primitive again and again is not an efficient approach. So Gelernter added some more primitives to the basic Linda model to avoid the multiple “rd” problems. In these primitives the tuple space itself is considered as objects i.e. the tuple space itself can be an element of a tuple. But this raised a few semantic problems for the basic Linda primitives such has \(<\text{out} (\text{tuplespace1}, 25\text{int}, 30\text{int})>\) and hence it is assumed that there are multiple tuple spaces incorporated in the existing Linda model.

The new primitives are Copy_Collect[2.6.1] and Collect[2.6.2]:

2.6.1 Copy_Collect

This primitive is used to solve the multiple read problems. The format of this primitive is \(\text{copy}_\text{collect} (\text{"source_tuplespace"}, \text{"destination_tuplespace"}, \text{template})\). This primitive works by copying the matching tuples with the given template from the source_tuplespace to the destination_tuplespace. It returns the number of tuples copied back to the calling process.

2.6.2 Collect

This primitive is similar to the copy_collect primitive. The format of this primitive is \(\text{collect} (\text{"source_tuplespace"}, \text{"destination_tuplespace"}, \text{template})\). This primitive moves the tuples destructively from the source_tuplespace to the destination_tuplespace. It also returns the number of tuples moved back to the calling process.

2.7 An Application - Mergesort

Although the reader has read and understood about tuple space in the previous sections, it is always better to explain using an example application in order to understand the
concept more practically. To understand the working of tuple space; an existing implementa-
tion is considered. The implementation is called as Tupleware Run-time system [12].

Any application can be considered provided that application can be parallelized. For
e.g. quick sort, matrix multiplication, etc. For my thesis work in order to get better
understanding, I have used the regular merge sort algorithm [13] and have implemented a
simple Merge Sort application. This application works on top of the existing Tupleware
system. This section clearly explains about storing the tuples in local or remote nodes,
searching for the tuples and retrieving the tuples from local or remote nodes. It also
provides an overview of how the parallel and distributed processing of tuples takes place
in an application. For more information about the Tupleware system please refer section
3.6. It also provides an overview of how the parallel and distributed processing of tuples
takes place in an application.

Any application can be considered provided that application can be parallelized. For
example quick sort, matrix multiplication, etc. The application considered in this case is
Merge Sort.

2.7.1 Description

This application works under the Master and Worker pattern. Any process or node can
act as master or worker. In general the working style is, the master node gives tasks to
the workers. Each workers gets these task, executes it and gives the result back to the
master. The workers do their work in parallel. The master collects all the results from
the workers and gives back to the application. The same working style is applied to the
merge sort application.

The goal is to sort an array of elements using the Tupleware Run-time system. In
this case the data is the unsorted elements. The data can be anything depending upon
the application. Let us consider there are 8 nodes and one million elements to be sorted.
One is the master node and seven are worker nodes. The master node initially divides the
array into some equal parts. Let’s say the master divides into 5 equal parts. So each part
has 20,000 elements. Each part is encapsulated in a tuple [“mergesort”, “unsorted”, ms].
The field “ms” is an object which holds the array of elements to be sorted. Now the
master randomly selects a worker node and post the tuple to it. Since it is a random
selection, there is a chance that not all the worker nodes will be utilized. For example
in this case there are five partitions. It is not necessary that those partitions have to
be sent to five workers, it can be sent to three workers such as worker 1 gets partition 1
and 3, worker 2 get partition 2 and worker 3 get partition 4 and 5. The master puts the
tuple directly into each worker’s local tuple space. Any data structures such as vectors,
hash tables . . . can be used for the tuple space. The data structure used in this example
is vectors. The storage criteria used here is a random selection. The master divided the
task and gave it to the worker nodes.

Simultaneously each worker is waiting for the partition using a template [“mergesort”,
“unsorted”, NULL]. Each worker gets a part of the partitioned array. The application
is designed in such a way that the worker nodes needs to communicate with each other
since it would benefit the search for a tuple in a remote node. The main processing loop
of the worker is evaluated again and again until each of them gets a “completed” tuple
from the master node. The following steps are performed by each worker concurrently
and parallelly.

1. Each worker node now has 20,000 elements. The worker performs two actions - split
or sort.
2. The workers split the array until they reach a threshold value. The threshold value in this case is 1,000 elements. The selection of the threshold value depends upon the application.

3. It then stores the remaining elements (i.e., 19,000 elements) in its own tuple space (i.e., in a vector or a hash table) or it can store in other worker’s tuple space. The decision depends upon the developer of the application.

4. Then the worker does a sequential merge sort for the 1,000 elements.

5. The worker now sends these sorted elements to the master’s local tuple space as \([ \text{"mergesort"}, \text{"sorted"}, ms] \).

6. If a worker has finished sorting its part (i.e., 20,000 elements), it now posts a template as \([ \text{"mergesort"}, \text{NULL}, \text{NULL}] \) to search for unsorted elements in other worker’s tuple space. In this way, the worker searches for a tuple which is not in the local node.

7. The template would match if there are any tuples in remote worker’s tuple space.

8. Again the steps 1 to 7 are repeated for this new set of unsorted elements.

Finally, the master is waiting for the tuple \([ \text{"mergesort"}, \text{"sorted"}, ms] \). It collects all the partitions and then re-constructs the array. It sends out the tuple \([ \text{"mergesort"}, \text{"completed"}, ms] \) to all the worker nodes to indicate that there are no more tasks to execute, hence all the worker nodes are terminated properly. The re-construction works well if there are less number of elements. But for larger number of elements such as for one million or more, it takes more time as only one process is working. It can also be optimised as, again the master can send the sorted partitions to each of the workers in order to merge them and give the merged array back to the master\(^7\). For detailed installation and source code refer appendix A.

### 2.8 Summary

This chapter summarised the background concepts such as the Tuples, Templates and Tuple Space. The example application would have given a better understanding for the reader. The reader is now assumed to have acquired the fundamental knowledge that is required as a pre-requisite to understand the following chapters.

---

\(^7\)The example is given for the purpose of understanding the basic tuple storage/retrieval operations and hence this optimization has not been implemented in my thesis.
Chapter 3

Analysis of Candidates

This chapter presents the analysis for some of the state of art tuple space implementations based on a set of criteria. Section 3.1 gives the details of all the problems which would occur when distributing tuple spaces. It also includes those conventional problems which would occur for any distributed application. The candidates are analyzed from Section 3.3 to Section 3.11. Each section describes the problems that are considered by the candidates and the solutions which were given by them to solve those problems.

3.1 Problem Statement

The Tuple Space has been explored outside Linda, in implementations such as Sun MicroSystem’s Java Space, IBM’s TSpace, York Kernel, etc. All these implementations have a centralized tuple space. This creates a set of problems such as single point of failure, overload of the tuple space as a single tuple space is being queried by different processes. Tuple Space coordination model is also suitable for distributed environment. It is very much similar to Distributed shared memory (DSM) architecture. In DSM, there are many nodes, each node has many processes and each process has its own memory. All the nodes are interconnected and communicate through the network. There can be various network topologies such as ring, mesh, star, bus, etc. A node can access other processors local memory only through the network.

The same analogy is suitable for Tuple Space. In a distributed environment, each node is considered to have its own tuple space. This tuple space is known as Local Tuple Space and every process can access (store/retrieve) their local tuple space using the Linda primitives which was described in section 2.5. All the nodes are interconnected through the network. The tuples which are stored in a remote node is known as Remote Tuple Space. This type of architecture is commonly called as Distributed Tuple Space by the coordination community. There are many implementations such as [4], [12] etc., which uses this architecture. In a distributed tuple space the tuples, (i.e. data) are stored in many nodes.

There can be many scenarios for a distributed environment as several nodes can be connected with-in the same room, with several departments of the same organisation, with organisation and third-party services residing at the same city and wide area networks with many cities in the same country. Our scenario for a distributed environment is almost similar to wide area networks but covers different geographical locations having millions of nodes. The figure 3.1 shows our distributed environmental context.

As the figure shows, the entities such as nodes, routers, firewalls, database etc, are familiar to any kind of architecture. The thick red lines indicate the connection between the geographical locations. To avoid complexity in the diagram a specific geographical
3.1. Problem Statement

Chapter 3. Analysis of Candidates

Figure 3.1: Our Distributed Environment context

location has been projected out. This architecture resembles more or less a peer-to-peer network.

Similar to any distributed system, in implementing a distributed tuple space, one faces a major set of problems that have to be resolved, such as:

1. How to store/distribute the tuples to the various tuple spaces (i.e. various nodes)?
   Consider there are five nodes N1 to N5. Each has their own tuple space. When a node performs an “out” operation, in which tuple space (node), the tuple should be stored.

2. How tuples be grouped and stored in the same tuple space (node)?

3. How can we efficiently retrieve a tuple which is not present in the local node? Suppose a process performs an “in” or “rd” operation, none of the tuples is matched with its own tuple space. Then the process has to efficiently search for the tuple in

14
remote node’s tuple space.

4. How do we achieve less bandwidth usage, less latency and less computational power? Since the tuples are distributed across several nodes, the computations i.e. the search and retrieval of the tuples should be efficient. There should be as little communication overhead as possible.

5. The traditional problems like scalability, fault-tolerance, load-balancing, adaptability are also applicable to a distributed tuple space.
   - Scalability: When the number of nodes is increasing or decreasing the system should be stable. Therefore the search for the tuple should be independent on the number of nodes in the system.
   - Fault-tolerant: When a node disconnects or crashes, care should be taken to those tuples which are present in those nodes.
   - Load-balancing: The data should be shared between the nodes. So that the nodes are balanced.
   - Adaptability: The system should be possible to reconfigure at runtime.

3.2 Eligibility Criteria

This section describes the various criterias that are essential for a candidate to be chosen. Hence the candidates should be able to satisfy them and prove that a distributed application can be implemented as a new run-time system or by extending the existing system. The below mentioned candidates are being analyzed based on the following criterias.

The system
   - should be using the Linda primitives or equivalent.
   - should store similar tuples in one or group of tuple spaces.
   - should provide primitives for bulk movement of tuples.
   - should provide primitives for synchronous and asynchronous operations.
   - should provide location transparency.
   - should provide storage and distribution of tuples evenly to all nodes.
   - should provide a decentralized and efficient search algorithm.
   - should provide a run-time system which is scalable and fault-tolerant.
   - should provide load-balancing using full-replication or partial replication strategies.
   - should be able to with-stand network failure, dynamic re-configuration.

3.3 York Kernel

The various implementations using tuple space started early in the year 1995 when A. Rowstron, Alan Wood [2] developed a kernel at the York University. It is known as the York Kernel. The kernel is like a controller for the tuple space. Their system aims for parallel execution between processes by exchanging tuples using the basic linda primitives. Each process has a tuple space which contains subset of tuples.
The tuples are distributed to various processes explicitly by the user using a hash algorithm. The hash algorithm selects the tuple space based on the number of fields and the type of the first field. i.e. process 1 stores only one field, process 2 stores two field and so on. This makes grouping of similar tuples possible. For instance all those tuples which are having three fields will be stored in a single tuple space. Examples of tuples used in their system are

- \((100_{\text{int}})\) and \((20_{\text{int}})\) which are stored in process 1 (p1) and type of tuple is single field,
- \((20_{\text{int}}, 5_{\text{int}})\) which is stored in process 2 (p2) and type of tuple is two fields,
- \((7_{\text{int}}, 25_{\text{int}}, 32_{\text{int}})\) which is stored in process 3 (p3) and type of tuple is three fields etc.

Since the user has to explicitly say to which process the tuple is sent, it increases the burden of the user.

The tuple retrieval is done by using the same hash algorithm (i.e. Each tuple or template is given as input to a hash function). The result of this function is the process which is going to store that tuple or template. So the user process asking for a tuple, can give the template to the hash function which will return the process that contains the tuple. For example, let us consider the user process is p1. It now sends a template \((20_{\text{int}}, 5_{\text{int}})\) to the hash function. The result would be then p2. Then the user process p1 can directly communicate to p2 for the tuple. This makes communication to be one to one.

The extension of this system was again given by A. Rowstron and Alan wood [3] in 1996. This time they focused on performance of the system by introducing bulk movement of the tuples from one tuple space to another tuple space. They have reduced the user’s burden by making their implementation to have an implicit approach. Their results have proved that sending multiple tuples in a single message across the network increases the performance of the system. Hence the bulk primitive such as “collect” and “copy-collect” seems worth.

3.3.1 Short conclusion

The system satisfies only a few criterias which are essential for a tuple space implementation such as tuple distribution, grouping similar tuples, tuple retrieval and bulk primitives. But the system still needs majority of criterias to be developed in order to change this system to suit the requirements. Hence this system is not suitable for a distributed environment. Moreover the implementation is mainly for parallel execution of processes rather than for a distributed system. Since each tuple space stores only one type of tuples, there is always a chance that tuple space can be used by many processes (i.e. causing bottleneck) or would not be used by any process. This method would also increase the use of many hardware resources (such as nodes) as each type of tuple requires a node. An efficient search algorithm needs to be implemented in this system. The internal data-structure used for tuples is a list. This would take more time while searching for the exact tuple because the search in a list is sequential. Instead of a list, a hashing tree can be used which would increase the speed. Since the authors have focused only on parallel execution the tuple search algorithm has to be extended to include remote tuple retrieval in order to fulfill the main requirements.
3.4 GSpace

Russello, Chaudron and Maarten in their paper [4], [5] say that there is a need for having different distribution patterns depending upon the type of data and the type of application used since it increases the performance of the system. Therefore using a single distributed pattern for all types of applications will not work as desired and we need to dynamically adapt to the various distribution policies. They are concerned with the problem of communication overhead between shared data spaces of various components in a ubiquitous environment. For this they have used tuple spaces to solve that problem by proposing a distributed architecture that separates the functional components from the data distribution. Their system is known as GSpace which was implemented in Java [6].

The GSpace system provides a communication middleware which is used by the in-home components. The important feature about the design of the system is that, the application developer can itself provide the suitable distribution pattern that would work for the application. Eventhough it would increase the burden of the application programmer the reasons they claimed in [5] for using different distribution strategies along with their results found seems to be quite interesting and acceptable. For example an application would like to have a persistent storage, another might need certain replication strategies. Hence, all these can be given by the developer at the begining and the system would work as desired. Thus they provide us a convenient way to use our own distribution strategies. For example,

- Strategies are:
  1. Store locally,
  2. Migrate-on-demand,
  3. Full replication,
  4. Push-to-all,
  5. Push-to-one,
  6. Push-where-needed,
  7. Cache with verification,

- Application usage patterns are:
  1. Local usage,
  2. Write-many usage,
  3. Read-mostly usage.

They have conducted many experiments using their system as which distribution strategy works well depending upon the application usage pattern. For example the “store locally” strategy would be more suitable to applications which have “local usage” application pattern.

They have also extended their architecture of GSpace to support the dynamic changing of the application behaviour. They focussed on adaptability, flexibility and bandwidth usage in [7] and availability [8] of distributed systems. To support these features, their GSpace system is extended to have an Adaptation sub-system. This sub-system is used to measure the cost that occurs due to the distribution strategy deployed by constantly monitoring and evaluating the system. The Cost function is calculated based on the average latency for “read” and “take” operations, memory consumptions for storing the
3.5. Grinda

Grinda is a tuple space system for large scale infrastructures which was given by Sirio Capizzi [9] in his PhD thesis. Grinda is a combination of Grid Technology and Linda. The problem of handling large number of tuples and distribution of tuples was his main area of research. He used a new approach to efficiently handle those problems.

To handle large number of tuples he had used Spatial Database techniques. Spatial indexing is used in many applications such as distributed databases, geographic information systems etc. This technique is used for querying objects from the spatial databases so that it can be effectively retrieved. For this he analyzed many spatial indexing methods such as X-Tree, Pyramid, KD-Tree and Quad Tree. To solve the problem of distribution of tuples, structured peer-to-peer(P2P) networks were used. The advantage of using the structured P2P is that these peers know where the resources are available and have an uniform routing. There are two types of structured P2P networks.

1. Distributed hash tables such as CAN[10]¹, Chord, Pastry[11] etc.

2. Tree based strategies such as VBI-tree etc.

---

¹Content Addressable Network
Peer-to-Peer systems are most widely used in many implementations since they are fault-tolerant and scalable. He has given both centralized and distributed prototypes for the tuple space implementation. The centralized is based on spatial index and distributed is based on structured peer-to-peer network. In both cases there are two phases: 1) A mapping phase which maps the tuples or templates to a n-dimensional space, 2) An indexing data structure for centralized or structured P2P network for distributed which does the insertion, lookup and removal operations. The P2P network should be able to satisfy the requirements of the spatial indexing such as to handle range queries since the “take”(in) and “read”(rd) operations are based on range queries.

These prototypes were then implemented on a grid provided by a open source tool called Globus Toolkit. The results obtained for his both forms of prototypes (centralized/distributed) seems to be good. But only the centralized prototype is used to build the grid service due to limited number of resources. The applications used to test the implementation are 1) highly parallel application hence it will have less communication overhead, 2) non-parallel application hence it will have more communication overhead.

3.5.1 Short Conclusion

After analyzing the Grinda tuple space system, it satisfies some of the requirements. But at the same time it cannot be neglected because it has lot of potential and scope to be developed to a fully distributed system. The system is also very much interesting as it uses spatial indexing techniques which have been mostly used in the database field till now. Spatial indexes are also used in distributed database applications, hence it will be more useful to retrieve objects efficiently when tuple spaces are distributed. The author Sirio have already provided a distributed tuple space implementation using structured P2P network such as CAN and VBI-tree but has not integrated into the Grid service. The Tuplebroker class in the server module is easily configurable which makes a developer to switch to distributed tuple space. To avoid system crash or service shutdowns persistent tuple space are used which uses XML databases. The associative matching rule has also changed in this system to be more efficient. This can be one of the most promising candidate to be chosen.

3.6 Tupleware

Alistair in 2008 gave a dynamic and decentralized search algorithm for the tuples in his paper [12], [14]. He called his system Tupleware [15] run-time system which uses a distributed tuple space and is implemented in Java. Scalability and latency are the problems which are addressed here when they distribute the tuple space.

Tupleware approaches these issues with a decentralized and intelligent tuple search. It provides an efficient way for the search and retrieval of tuples. Since it uses a decentralized search algorithm it reduces the burden of the programmer. Each node has a Tupleware run-time system which has a stub, service and tuple space. The stub and service are used for the communication between the nodes. Here the tuple space is implemented using dynamic arrays in Java known as vectors. There can be many worker nodes which are trying to connect to the master node. So the service in the master node creates separate threads for each worker that would handle the corresponding request from them.

They attempt to store the tuples locally from the producer end and then use their intelligent search to retrieve the tuple remotely from the consumer end. The search algorithm works in such a way that there is no change in the retrieval of tuples from the local tuple space. But the retrieval of tuples from the remote tuple space depends
upon the “success factor” of the corresponding node. Initially the success factor for all
the nodes would be set as 0.5. If a node satisfies the request then its success factor would
be increased and move towards 1. If a node did not satisfy the request then its success
factor would be decreased. The workers communicate with each other in a decentralized
fashion i.e it is not necessary to communicate with the master. This is similar to the
fitness function described in Swarm Linda \cite{23} system.

Each node has a sorted array of stubs hence in the case of a remote search, the
stubs will be searched in the order of the array. After every search by each worker the
array would be sorted again and the stub whose success factor is more would be the first
element in the array. Due to this the search algorithm adapts dynamically by the changing
communication patterns and in some time it will have optimal tuple search. For example
if a node does not match a tuple then automatically its success factor would decrease and
then its stub would be placed in the last position of the array. The Tupleware is designed
to suit well for array based applications. The distributed tuple space is transparent and
looks like a centralized system from the developer point of view and hence any parallel
application can be easily developed on top on their run-time system.

3.6.1 Short conclusion

The analysis of this candidate seems to be quite good. It is an hybrid system (cen-
tralized/decentralized) depending upon the application. He has pretty much given a
good solution through their decentralized search algorithm. But the only disadvantage
is that the alogorithm is application specific. Hence the search algorithm can be impro-
vised. They provide primitives for the bulk and asynchronous operations. If fault-tolerant
strategies are added and load balancing strategies are improved, then this system would
be more suitable for our distributed application. Since they have developed a distributed
tuple space it is not necessary to follow a master/worker pattern. Rather we can have
a peer-to-peer system. In that case it would be a geographically decentralized system.
Hence this is one of the most promising candidates.

3.7 DepSpace

Most of the candidates which were analyzed so far have been focussing on issues such
as scalabiltity, remote search, openness etc. One of the important requirements for any
distributed sytem is that the system should be reliable in the event of network failures,
intrusion from malicious users, server crashes etc. Bessani, Alchieri, Correia and Fraga
\cite{16} in 2008 focussed on these issues and gave a system called as “DepSpace - A Depend-
able tuple space” implemented using the Java programming language. A system is said
to be dependable when it satisfies the following properties such as reliability, avaialability,
integrity and confidentiality inspite of the occurence of Byzantine failures\textsuperscript{2}. To make the
system to be fault-tolerant they have used State Machine Replication approach. There-
by it ensures reliability, avaialability, integrity but NOT confidentiality. This approach
enforces a strong consistency between the coordinating processess. The replication is
supported by using totally ordered broadcast\textsuperscript{3} protocol. There are also other ways to
handle fault-tolerance such as the one given by LiPS Runtime System \cite{17}. They also use
replication to make the system to be fault-tolerant but the method is different. In LiPS
system, every machine has to update their state information to the System tuple space\textsuperscript{4}.

\textsuperscript{2}It refers to the Byzantine General problem.
\textsuperscript{3}All processes maintains the same state.
\textsuperscript{4}A tuple space which is shared by all nodes.
in frequent intervals of time. So if a system has not reported its state is considered to be crashed or failure.

Confidentiality was bit tricky to implement using tuple space model because the normal way is to share a private or public key which is known to both client and server. But this will contradict the space decoupled property of tuple space model. To overcome this limitation they have used Publicly verifiable secret sharing scheme (PVSS) which provides confidentiality to their system. They have integrated confidentiality into their system by encrypting the data. There by their system is considered to be a secure system.

3.7.1 Short Conclusion

From the analysis I came to know about some good things about their system. They have provided good solutions for security and fault-tolerance. It was the first system at that time to provide a solution which combines both the security and fault-tolerant features. These kind of systems are essential in a distributed environment as the environment is considered to be unreliable, dynamic and untrusted. They have given both synchronous and asynchronous primitives in their implementation. A new primitive called Conditional Atomic Swap (CAP) is also given, to solve the consensus problem which normally arises due to replication of the data. At the same time the bulk primitives are not provided. But it can be considered as a weak critiria in this case since they focussed on security and fault-tolerance of a distributed system in which there might not be bulk movement of tuples. It is a centralized system that is replicated to tolerate faults and ensures strong consistency. To satisfy other requirements this system needs to implement a decentralized search and focus on scalability factor.

3.8 DTuples

The system which was given by Jiang, Jia and Jinyuan [18] is called DTuples. It is a tuple space implementation using a distributed hash table and structured peer-to-peer network. They haven’t focused on a particular problem but they wanted to develop a system which uses tuple spaces for peer-to-peer computing and extend that system for high performance computing [20]. There by they internally focus on issues such as fault-tolerance, tuple distribution and node failures. They did not focus much on scalability since peer-to-peer systems have the scalability feature in-built. The tuple distribution is based on distributed hash table which uses FreePastry[19] as the overlay network. The tuples in the DTuples system should follow a naming standard such as the first field should be of type “string”. This field is the name of the tuple. Some examples of Dtuples are

- (“t1”, 1),
- (“t2”, 3, 4.5, 72),
- (“t3”, 5.6, 15.52),
- (“t4”, “hash”, 4.5, 72),
- (“t5”, “overlay”, “network”) etc.

These tuples are then given to a hash function. The hash function takes the name as the key and then returns the node, to which it is then stored. For example
The tuples which have the same name would be stored in the same node. Since they are using a distributed hash table the tuples gets distributed in a fairly equal manner to all the nodes. Thus the load gets balanced between the nodes. But there would be problem when there are large number of tuples of the same name, in that case only one node will be utilized. They also have common tuple space (public) and subject tuple space. All the node agents can access the common one where as the subject tuple space can be accessed only by those node agents which are bounded to it. This is very much similar to the tuple space description of the Tupleware system in section 3.6 which has local and remote tuple spaces but the major difference is that more than one node can share the subject tuple space by obtaining permission from the owner.

Using the subject tuple space they provide the replication mechanism in their system which is essential for a system to handle node failures. They follow state machine approach similar to DepSpace in section 3.7 to maintain the replicas and ensure availability of the system. Tuple search is also carried out using the distributed hash table. The Templates would provide the name of the tuple to the hash function, which would return the node where the tuple is stored. There by it follows a simple routing mechanism in a decentralized fashion. For example: the templates would be

- \( (t1\), NULL),
- \( (t2\), 3, NULL, 72),
- \( (t4\), “hash”, 4.5, NULL),
- \( (t5\), NULL, NULL) etc.

A node can have both common and subject tuple space. Templates are first searched in the subject tuple space of the nodes to which they are bounded. If they are not found then they use the hash function to find the node in which the tuple was stored. Hence they follow a decentralized search pattern as the search for the tuple is not dependent on any other factors.

### 3.8.1 Short Conclusion

We have already analysed a system called Grinda 3.5 which also uses distributed hash table and structured peer-to-peer network. This system differs slightly in the overlay network that was chosen for the distributed hash table. Grinda chose CAN as the overlay network as it was easy to extend in order to use multiple range queries which were essential for their system. Here they have used Pastry overlay network for their implementation. Using Peer-to-peer network they have built their system to be decentralized. The problem is when tuples belong to the same template would have the same hash key and would be stored at the same location. But this can be solved by storing them in different nodes through formation of node clusters as in SwarmLinda in section 3.9. So that a group of nodes would store the tuples of the same name. Another highlight about their system which is worth mentioning is; this was the first system which implemented the “eval” primitive for active tuples. This system satisfies most of our requirements. This can also be a good candidate.
3.9 SwarmLinda

A new approach to solve the scalability problem in distributed systems is given by Ronaldo, M and Robert, T [21],[22], [23] in 2003 through their coordination platform called as SwarmLinda which is implemented in Java. They have used Swarm Intelligence\(^5\) techniques for solving the scalability problem. They have also focussed on distribution of tuples and adaptability in distributed systems. In nature, we can see that millions of ants, schools of fishes and flock of birds coordinate themselves by having a local interaction with their neighbours irrespective of their size. This motivated the authors to use this coordination mechanism of the ants in a coordination model such as the tuple space.

Usually when ants search for food, they start randomly from their ant hill. After an ant discovers the food, it puts a chemical substance (Pheromones)\(^6\) along its return path and reaches the ant hill. Other ants senses this chemical substance and reach the location of the food. Thus in course of time a path is created between the location of food and the ant hill. There is need to map the terminologies of the biological world to that of tuple space context. The “world” is mapped to a two dimensional set of tuple space servers. The “food” is the tuple and the “ants” are the templates which goes in search for food. They are called template-ants. The “ant hill” is the client process. Tuples are represented as XML\(^6\) documents in this system. The search algorithm given by them is more adaptable to suit the environment of the distributed system. A common problem faced by systems such as Grinda 3.5 and Dtuples 3.8 is the tuples having the same template would be stored in the same location since they use distributed hash tables. This is solved in this system by the formation of node clusters. So a group of nodes would store the same type of tuples.

Brood sorting strategy is used for tuple distribution in this system. Brood sorting is performed by ants which is used to sort based on type such as larvae, eggs . . . irrespective of the size of each type. Here the mapping is the same similar to search algorithm discussed above but the “ants” are those which are going to store the tuples. They are called tuple-ants. Each node maintains its own list of neighbours. This list is updated through a fitness function of a node. This fitness function depends upon factors such as node liveliness and tuple matching of that node. By this the adaptiveness of the system is obtained.

3.9.1 Short conclusion

Efficient algorithms are given for search for remote tuples, openness and distributing tuples equally. These algorithms ensures adaptability, scalability, fault-tolerant and load balancing of the system. Scalability is achieved in the sense that the decisions taken by the ants are local by communicating only with the neighbourhood. So even when the number of systems scale up or down, does not have an impact in the search algorithm. Adaptability in the sense the decisions are taken from the current state of the system. The state includes the kind of tuples produced and consumed. Each node further maintains a neighbour list similar to routing tables in networks to have up-to-date information. This further supports the adaptiveness of the system as each nodes is aware of whether other nodes are working or down due to network or server failure. Due to this adaptability the swarmlinda system is fault-tolerant and load balanced. This candidate satisfies most of the requirements. It is also very much interesting to know as this candidate is unique because they have used Swarm Intelligence techniques and solved many problems of a

\(^5\)A theory in Biology which refers to naturally coordinating species such as birds, fishes.
\(^6\)eXtensible Markup Language
distributed system. But the disadvantage of the system is that only a simulated Swarm Linda system is available [24] but the source code has not been published.

3.10 Lime

Tuple space is also used in mobile environment for developing mobile applications. A very good example for this is the LIME\(^7\) system developed by Murphy, Picco and Roman in 1999 [26], [27]. There are two components in the Lime System. They are mobile host and mobile agents. The authors consider mobile host to be laptop and PDA’s and mobile agent to be processes. The Lime system can be used for the typical mobile environment. Each mobile agent has its own tuple space known as Interface tuple space (ITS). The ITS’s are bounded to the mobile agents. As far as the Lime system is considered they don’t have a globally, shared persistent tuple space which is the original definition provided by Linda for tuple space. This is because in a mobile environment, the mobile agents will be moving in and out of range of the mobile host. Hence the use of persistent tuple space is not suitable in this context. Instead they define transiently shared tuple space. For example, consider there are two mobile agents as A and B. Both A and B have their ITS’s. The two agents are said to be connected if they are in the same range of the mobile host. From the view of the agents they would be seeing a merged/combined ITS’s. This is known as “Host tuple space”. In this way agent A can access the tuples from agent B’s tuple space. This tuple space is in transition because when agent A moves out of range, it will move along with its tuple space. Likewise many agents can join or leave the range of the mobile host. Joining is known as Engagement and leaving is known as Disengagement. The transiently shared tuple space is not only for mobile agents but can be extended to mobile hosts as well. In this case the two mobile host are said to be connected through a physical link. Similarly from the view of the agents they would be seeing a combined host tuple space. This is known as “Federated tuple space”.

3.10.1 Short Conclusion

The system is mainly developed for mobile environments which is mostly not the major aim of the thesis. The original semantics for tuple space have also been changed due to the mobile environment. So considering this factor to be a major drawback, this system cannot be considered for further study.

3.11 Other Candidates

3.11.1 World wide Coordination Language

A.Rowstron and Pembroke Street in 1998 [28] developed the first run-time system which is geographically distributed using tuple spaces. They have not used the linda primitives for building their system. They have developed a new set of primitives which aims for efficiency and location transparency. This is known as WCL - World wide Coordination language. The run-time system consists of three parts. They are Tuple management system, Control System and the libraries. The tuple management system is used to manage the tuple servers. Each tuple server has a set of tuple spaces. Each tuple space is identified by a unique handle. The control system is used for server and agent registration. Location transparency is achieved since the tuple server is registered by the

\(^7\)Linda in Mobile Environment
control system. There by the location of the tuple space is known which is then used by
the agents for searching for tuple space.

Even though they have some advantages such as geographically distributed run-time
system, bulk and asynchronous access primitives, language transparency, load balancing
of the tuple servers and efficient primitives equivalent to the linda primitives but they
lack in basics. Since their implementation is mainly focussed on tuple spaces rather than
on individual tuples. This made me to think whether to choose this candidate or not.
The first reason is that the agents instead of searching for tuples in remote locations they
search for tuple spaces. The second reason is instead of migration of tuples they move
tuple spaces from one tuple server to another there by increasing the load on the server.
Due to these reasons this candidate is not chosen. There were many critizes to Rowstron
as his work is more coarse grained and does not deal with the individual tuples rather it
deals with tuple spaces.

3.11.2 uTupleSpace

A fine example where tuple space is used in Ubiquitous/Pervasive environment is given
by Takayuki, Hiroya [32] through their implementation of uTupleSpace system. Here
the tuple space model is used in connecting various sensors and actuators in a WAUN8.
They focus into problems such as scalability, load-balancing, common data format for
communication between devices and openness. To have a common format they have
modified the original tuple as uTuple (metadata + tuple) to suit the environment. This
data format is used for communication between the devices. The metadata includes
address, time, position, subject and type of data. They have included the address of the
device as it is essential to be known for this environment. But this violates the space
decoupled strength of the original tuple space model. The uTuple format can be used
only by this environment which is a limiting factor. They have named their tuple space
as uTupleSpace.

Scalability and load-balancing problem is solved by using distributed hash tables.
They have used two phase matching scheme; 1) Finds the server using DHT, 2) Finds
the uTuple using RDBMS9. Their design is more similar to the Grinda system described
in section 3.5. The server will be fully loaded in the sense when different uTuples get the
same key. This is the same problem which was not solved by the Grinda 3.5 and Dtuple
3.8 systems. But this system provides a solution by having a dynamic scaling method
[33].

3.11.3 JavaSpace

JavaSpace [34] is an industry implementation on tuple space developed by SUN devel-
oper network. It uses the power of Java language and it is centralized system. Many
implementations have been developed using Java space such as Gigaspaces and TSpace.
It provides transaction support which is not present in TSpace. Since it is not a decen-
tralized system, it has not been choosen for further study.

3.11.4 T Space

IBM also contributed to the implementation of tuple space at their Almaden Research
Center in 1999. Lehman from IBM, McLaughry and Wyckoff developed a middleware
called as “T Spaces - Intelligent Connectionware” [29] using tuple spaces. They focussed

---

8Wide Area Ubiquitous Network.
9Relational Database Management System.
on the communication between heterogeneous systems in a ubiquitous environment. To solve the varying communication needs between different systems they proposed a new middleware which is a combination of tuple space, database and Java. Since they include database, their system was able to provide stability, durability, file and transaction features which are important to any kind of database system. Even though T Space is used for distributed applications, it is a centralized system. Tuples are retrieved from the central tuple space through database indexing mechanism using queries. Server failure is handled by maintaining log files so that when the server fails, all those transactions before the failure can be re-invoked. Applications which are distributed and needs data storage can use these type of systems. T Space version 3 [30] has much more advanced features which leads to a robust and secure system.

I would like to say that T Space system is in its early stages. The Fault tolerant support through replication is in development process in the version 3 of the system. Moreover the T Space system is not an open source implementation and it is closely used by IBM and projects which are related to IBM. For example OptimalGrid project [31]. So considering this fact and also many features such as decentralized system, remote tuple search, scalability … are lacking and hence this system is not considered.

### 3.11.5 Gigaspaces

Gigaspaces [35] is an implementation of tuple space on the basis of Java Space technology. It is developed by Gigaspace Technology Limited\(^\text{10}\). It is distributed and provides linear scalability along with fault-tolerant features. It is the first commercialized system which is developed for SBA\(^\text{11}\). Clustering is also provided through the virtualization of the middleware. This is a good candidate to be chosen.

### 3.12 Summary

This chapter would have given the knowledge to the reader about the analysis of the state of art tuple space implementations which were used in many applications. This chapter also provided brief details about each system and motivated the reasons behind each system’s pros and cons. We can see the evaluation of these candidates in the next chapter.

\(^{10}\text{www.gigaspaces.com as on December 19,2011.}\)

\(^{11}\text{Space Based Architecture}\)
Chapter 4

Evaluation and Extension of Candidates

This chapter explains the criteria’s used in my thesis for evaluating the various candidates. The definition of the various criteria’s is given in section 4.1. The analysis report is given in a matrix tabular form in figure 4.1. Section 4.3 gives the discussion part of my thesis. It gives the reasons for selecting those criteria’s along with the weigtage of each criteria in the respective groups. Next it states the reasons for filtering the candidates and also gives the reasons why there is need to propose a novel solution? The possible extensions and ideas are given in section 4.4.

4.1 Description of the Criteria’s

Linda Primitives or Equivalent: This criteria filters those systems which uses the basic Linda primitives or any other primitives which are equivalent to them. So if any system uses them, then it is said to satisfy this criteria.

Similar Tuples/ Clusters: This criteria specifies whether grouping of similar tuples are provided by the system or not? Grouping is essential and can be done in many ways. For instance, tuples belonging to the same type and having same number of fields can form a group. For example:

- Group 1 - Integer with 1 field
  1. (4)
  2. (56)
  3. (24) etc.

- Group 2 - String with any number of fields
  1. (linda, program)
  2. (parallel)
  3. (distributed, cloud, internet, java) etc.

- Group 3 - Float, String, Integer, String with four fields
  1. (54.7, house, 5, tuples)
  2. (74.2, string, 8, template)
  3. (23.2, hash, 9, node) etc.

Likewise any number of groups can be formed. In case of larger number of tuples, a single node will not be capable of storing them due to other nodes accessing this
node, bottleneck problems and the load of the tuples will not be properly balanced. Hence those tuples can be stored to the nearby nodes forming cluster of nodes of the same type. In my view, those systems that have the formation of clusters rather than grouping of tuples would be more advantageous and would be given more weight-age.

**Bulk Primitives:** This criteria states whether bulk primitives are given by the system or not? Using Linda primitives only a single tuple can be stored or removed at a time. i.e. when two tuples match the template, only one would be retrieved non-deterministically. Hence, there is always a need to retrieve or store many tuples which matches the template to increase the performance of the system.

**Asynchronous Primitives:** This criteria states which systems provides the non-blocking primitives such as INP and RDP. They are essential because a process cannot be waiting to read or retrieve a tuple which is not present in the tuple space. Hence, instead of waiting, a NULL value could be returned.

**Location Transparency:** It defines that the location of the tuples is transparent to the clients. The client interacts with the system in the same way as the remote client.

**Decentralized System:** The systems should not have a central control like client-server model. Peer-to-peer systems are mostly preferable. Any decision concerning the system should not be taken by an individual instead any individual should have the authority to take decisions.

**Distribution of Tuples:** Tuples can be distributed to a single node or to a group of nodes (clusters) based on the type using some techniques such as DHT, Tree etc. This criteria specifies whether the system uses any techniques or novel approaches to distribute tuples to all the nodes.

**Decentralized Search:** It involves the algorithms that are used to retrieve the tuples which are not present in the local node. The algorithms given by the system should also behave independently to the number of systems. Hence the algorithms stated by the different systems should take decisions locally and should not have a global view.

**Scalable System:** This criteria denotes whether the system is scalable or not? Scalability factor is more important for any kind of distributed system. Scalability depends upon the number of nodes and the application used. Sometimes the number of nodes would be increasing or decreasing and sometimes the application would be scalable. A system satisfying both these dependability’s would have more weight-age.

**Fault-tolerant System:** Any distributed system should be designed to be available whenever the client is requesting. i.e it should be fault-tolerant. It can be achieved in many ways such as replication, partial replication, intermidiate replication. There should be a consistency mechanism which is used to maintain the replicas in several servers. So this criteria is used to filter those candidates which uses fault-tolerant mechanisms.

**Load balancing:** This criteria denotes whether the system is balanced or not. When a single node is filled with tuples, the run-time system provided by the candidates should automatically balance by having some mechanism to shift the tuples to other nodes. Similarly when there are very less number of tuples in a node, the system should shift to other nodes from this one. There by using the system resources effectively.
Reliability: This is essential for any distributed system. Reliability ensures availability, confidentiality, integrity. It specifies whether the run-time system supports network and server failures and the methods to overcome those failures.

Dynamic re-configuration: This specifies whether the run-time system can be dynamically configured? It is the openness or the dynamism which is important for a distributed system.

Security: Any application or any system should be secure. There can be several attacks such as intrusions in the network, byzantine attacks, authentication problems. This criteria specifies which system provides the basic or the advanced security features.

4.2 Analysis Report

See the Tabular form in the following page.
<table>
<thead>
<tr>
<th>Criteria’s \ Systems</th>
<th>Java Space</th>
<th>Tspace</th>
<th>Giga Space</th>
<th>York Kernel</th>
<th>Wd</th>
<th>Dtuples</th>
<th>Grinda</th>
<th>Tupleware</th>
<th>Dep Space</th>
<th>Swarm Linda</th>
<th>uTuple Space</th>
<th>GSpace</th>
<th>Lime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linda Primitives or Equivalent</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Similar Tuples or Clusters</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bulk Primitives</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Asynchronous Primitives</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Location Transparency</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Decentralized System</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Distribution of Tuples</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Decentralized Search</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Scalable System</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fault-tolerant System</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Load balancing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Network/Server Failure (reliable)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic re-configuration</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Security</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 4.1: Analysis Report
4.3 Discussion

The criteria’s have been grouped under three major categories. They have been clearly distinguished with their color. The groups are:

Tuple space criteria’s The requirements in the first group (blue color) symbolizes that these requirements are commonly used for tuple space implementations. They cannot be seen for any other implementations. The reason for selecting them because, in most of the candidates which I analyzed were using one or most of them in their respective proposals and implementations. So I also wanted to have them for evaluating my system. These criteria’s would be good as my thesis deals with tuple space. The weight-age or the importance for this group also varies as they have a descending order\(^1\).

Thesis criteria’s The next group (red color) is more specific towards my thesis goals. The reason for selecting them because, to solve our distributed environment scenario which is explained in section 3.1, these criteria’s are useful. They are the most important criteria’s which has to be satisfied. The “decentralized system” criteria has the highest weightage than the others in this group.

Distributed system criteria’s The last group (dark red color) symbolizes that these requirements are commonly used for any distributed application. Again the reason for selecting them comes from the candidates implementations. The weightage for this group is equally shared.

The systems which were analyzed are also grouped. The first three systems (dark blue color) are those given by the industrial arena. The fourth system (brown color) is unique as this is the first one developed by Gelernter using tuple space. The next six systems (light blue color) are open source, small scale projects developed as university thesis. It involves LAN\(^2\) and WAN\(^3\) implementations of tuple space. The last three systems (green color) are based on tuple space implementations in ubiquitous and mobile computing fields. The region marked in red dotted lines is the one which is more interesting and the explanation is as follows.

The six contestants were choosen because the implementations for these were based on local and wide area networks for a distributed system. They were also much suitable and desirable for our distributed environmental context explained in section 3.1. Another reason is that they are open source implementations. So in case, if I wanted to extend them these would be more useful than the rest. Even though the last group is quite a good choice since the uTuplespace or GSpace systems satisfies much of the criterias and also provide open source implementations, they were neglected since their work is based on ubiquitous and mobile computing fields. So from 13 systems, it got converged to 6 systems.

The satisfaction table for the six systems is given in table 4.1. This table gives the idea about how many criteria’s are satisfied in each group by the candidates.

Among the six in the group; Swarm Linda, Dtuples, Tupleware and Grinda got the first, second and third positions respectively. Wcl and DepSpace systems were neglected because they did not satisfy the most important requirements in group 2. Dtuples and Swarm Linda were choosen because these candidates satisfy the majority of the criterias in each group and as a whole which can be seen from the table. Since the third position

---

\(^1\)The most important is the first one and the least important is the last.
\(^2\)Local Area Network.
\(^3\)Wide Area Network.
4.4 Extentions and Ideas

Chapter 4. Evaluation and Extension of Candidates

<table>
<thead>
<tr>
<th>Group No/Systems</th>
<th>Wcl</th>
<th>Dtuples</th>
<th>Grinda</th>
<th>Tupleware</th>
<th>DepSpace</th>
<th>SwarmLinda</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Tuplespace implementations</td>
<td>4/5</td>
<td>3/5</td>
<td>3/5</td>
<td>4/5</td>
<td>2/5</td>
<td>4/5</td>
</tr>
<tr>
<td>2. Thesis criteria’s</td>
<td>1/3</td>
<td>3/3</td>
<td>2/3</td>
<td>3/3</td>
<td>0/3</td>
<td>3/3</td>
</tr>
<tr>
<td>3. Distributed application</td>
<td>2/6</td>
<td>4/6</td>
<td>3/6</td>
<td>1/6</td>
<td>3/6</td>
<td>5/6</td>
</tr>
<tr>
<td>Total</td>
<td>7/14</td>
<td>10/14</td>
<td>8/14</td>
<td>8/14</td>
<td>5/14</td>
<td>12/14</td>
</tr>
</tbody>
</table>

Table 4.1: Satisfaction Table

was a tie between two systems, I had to make a choice to select one. I choose Tupleware system because it scored higher in the individual groups than the Grinda system. Now we can see that from 6 systems, it got converged to 3 systems which have the potential to be developed/extended to a distributed system satisfying our distributed environmental context.

4.4 Extentions and Ideas

There are three possible ways and each one is explained below:

1. Extention of Swarm Linda to have Security 4.4.1.
2. Extend Tupleware system to be fault-tolerant 4.4.2.
3. Propose my own solution 4.4.3.

4.4.1 Swarm Linda with Security

From the beginning Swarm Linda was quite interesting in their approach which was used to solve the problems of distributed tuple space. They had attacked the problems using Swarm intelligence techniques. The two criterias which was not satisfied by this system were Bulk primitives and Security. Among these I thought, a Secured Swarm Linda would have more importance. Hence wanted to extend Swarm Linda to have security features. From this paper [25] it ensures that though proper and efficient algorithms were given for the tuple-storage, tuple-retrieval, ageing mechanism for the ants, separation of concerns, multiple tuple spaces . . . , but only few of them were implemented in the initial prototypic implementation. In this prototype they have assumed the network topology of Swarm Linda to be similar to a peer-to-peer network. The final Swarm Linda system was implemented in a simulated environment using NetLogo\(^4\) but unfortunately the source code has not been published to the outside world. Hence this constraint dissuaged me from extending Swarm Linda to support all features.

4.4.2 Fault-tolerant Tupleware

There are many features in which a Tupleware system can be extended. From the satisfaction table we can infer that it needs to satisfy location transparency in group 1 and almost all features in group 3. Location transparency can be considered as least choice and hence extending one or all the features in group 3 is taken into consideration. Figure 4.2 shows the current Tupleware architecture.

The run-time system from the figure above deals with the logic/algorithm for the tuple search and retrieval. The algorithm is quite simple and is application specific. There are two possible ways to extend Tupleware.

\(^4\) Used to generate network simulation environments.
The first one is from the brief conclusion which I gave for this system in section 3.6. So as I said, it would be better if the algorithm used in Tupleware can be modified as the current algorithm is application specific. For this I have chosen the algorithms given by Swarm Linda system to be used for this extension. There are three specific reasons to choose Swarm Linda algorithms.

- The algorithms have the advantage of dynamic adaptation which can be used for optimization of distribution of tuple retrieval.
- The algorithms are self-organising and hence there is no need to program clustering of nodes by a developer.
- The probability terms used in these algorithms makes sure that the system explores much, when more nodes are added. Hence it is not constrained to a particular set of network topology.

By having this algorithm we can ensure the system supports dynamic re-configuration on a more coarser level. The cost of changing the algorithm involves the changing the run-time system itself. This also involves changing the underlying tuple and template structures which as for now (currently) is considered as objects of the tuple and template classes. This extension is quite costly as it requires a change in the run-time system and then add features to support fault-tolerance, load balancing . . . .

A better way (second way) is to keep the algorithm as such but extend the run-time system to support these features. A Node manager can be added to the current run-time system as an extension which is shown in figure 4.3

The functionality of the node manager is that each node has to maintain a dynamic neighbour list which is used to monitor the nodes which are up or down. This is similar to how routing tables in the networks are up-to-date with their routing information. The
connectivity for the nodes can be flexible in case any number of nodes can be neighbours for a particular node or can be fixed in case only a pre-defined number of nodes can be neighbours for a particular node. Having the node manager feature enabled we can provide the reliability feature and adaptiveness to this system. To add fault-tolerant feature, we can follow the techniques which were used by DepSpace and Dtuple systems. As it was described in their respective sections 3.7, 3.8, they have State machine approach for replication. But if we adopt this approach, it would not result in a distributed tuple space which will eventually change the major requirements.

4.4.3 Own Idea

The proposal of a new approach which solves most of the problems is given in the following chapter.

4.5 Summary

This chapter would have given the evaluation of the candidates and possible extentions. In this chapter the readers would have understood the filtering process of the candidates. i.e., from 13 systems to the final 3. The next chapter deals with the design/architecture of a new solution which was evolved due to my analysis and evaluation.
Chapter 5

System Design/ Architecture of the Proposed Solution

The chapter explains the design and architecture of the proposed solution. This chapter begins with the power of Erlang language in section 5.1. How tuples and templates can be represented in Erlang is given in section 5.2. The design requirements are given in section 5.3. The architecture of the solution is explained in section 5.4 followed by the explanation of individual components of the system in section 5.5. The data flow and the communication flow of the system is given in section 5.6. The satisfaction table for the new proposal is given in section 5.7.

5.1 Choice of Language

Erlang [36] was developed at Ericsson as a proprietary language in the mid 1980’s and later it was released to the open source community. Erlang was developed for achieving concurrency, fault-tolerance and robustness in massively parallel and distributed systems. It is well supported along with the Open Telecom Platform (OTP) which provides a standard for building telecom applications with its tools and libraries. Process creation, fault-tolerance and communication between processes are trivial tasks in Erlang. Most of the candidates analyzed in chapter 3 have been implemented in Java. There were no implementations using Erlang. So one of the reasons for choosing Erlang rather than Java is that the proposal should have a novel approach right from the beginning stage and also use the advantage of Erlang for my design and implementation of the solution. The efficiency of Erlang processes and the lightweight nature of its message passing made it easy to spawn processes in order to handle request and to build using OTP.

5.2 Tuple and Template Representation in Erlang

Tuples are nothing but ordered list of elements. Each element has field and value pair. Examples of tuples are:

- t1 (4_int, linda_string, 10.65_float),
- t2 (parallel_string, cloud_string),
- t3 (25_int, 100_int, 99_int) etc.

Representing them in Erlang was a much easier task since Erlang has a “tuple” data type. It is a composite data type which can hold any number of fixed elements. It is been delimited by curly brackets.
The above tuples are represented in Erlang as:

- \{4, linda, 10.65\},
- \{parallel, cloud\},
- \{25, 100, 99\} etc.

Templates have the same definition as tuples with the inclusion of NULL values. Examples of templates are:

- temp1 (10\_int, NULL),
- temp2 (book\_string, NULL, NULL),
- temp3 (NULL, 25\_int, 30\_int, 300\_int, NULL),
- temp4 (NULL, NULL, lan\_string, parallel\_string) etc.

The NULL values can be represented using “variable_name” in Erlang. The variables will hold the actual values during the matching process. The above templates are represented in Erlang as:

- \{10,Value\},
- \{book,Val1,Val2\},
- \{Val3,25,30,300,Val4\},
- \{Val5,Val6,lan,parallel\} etc.

### 5.3 Design Requirements

The following design rules have been formulated by me while designing the proposal. These are given in order to avoid the flaws which were present in some of the candidate solutions and also to satisfy the criteria’s in section ??.

- The design should have decentralized network topology. Example: peer-to-peer architecture.
- The design should not be application specific and should be simple.
- The design should have distinct components so that it can be easy to modify them or add new components in future.
- The system should have security features during the transmission of data over the network.
- Node connectivity, latency and bandwidth should also be considered while designing the architecture.
- Since we are dealing with geographically distributed system, the hardware capability of the nodes should also be considered. This would have an added advantage in the case of balancing the load.
5.4 System Architecture

5.4.1 Overview of the Entire System

Peer-to-peer architecture was the pre-dominant choice while designing the system mainly for its decentralized nature and for its in-built advantages such as self-organizing, adaptation and fault-tolerancy. They can be broadly classified into Structured and Un-Structured p2p networks. Un-Structured networks are further classified as Pure, Hybrid and Centralized p2p systems. The type of p2p architecture that is more suitable to build a distributed system across geographical locations and will satisfy the requirements is Super Peer Architecture [37]. I choose this type of architecture because there are some limitations in the regular p2p systems. For example in the Gnutella network which is an un-structured pure p2p system, they have lot of query messages which cause the entire network to be flooded. In the Napster network, the cost of having a centralized index is very high and also it has scalability and performance problems.

The overall view of the system can be seen from the figure 5.2. Each cluster denotes a specific geographical location which has a super peer node. A super peer node acts like a servant since it satisfies the request of the peer nodes and also forwards the queries to other super peers. Hence this type of architecture has both the advantages i.e. It has

---

37 A node which can function both as a client and as a server.
efficiency of a centralized system for lookup and the advantage of a distributed system
for having features such as robustness and load balancing. The super peer nodes are
connected with each other in a peer-to-peer fashion. Any p2p system is scalable and
provide fault-tolerancy. Hence the scalability and fault-tolerancy of the peer nodes or
super peer nodes is taken care from the design stage onwards.

In the design of conventional p2p systems like Gnutella all the peers are equipotent\(^2\). This is a disadvantage as distributed systems are heterogeneous\(^3\). This architecture can
be used to overcome the limitation since the super peer network considers the hardware
capability of the nodes. Hence the nodes which can handle many servers and clients can
be a super peer node. Other nodes can be a regular peer node having less hardware
capabilities. Any routing protocols such as Chord, Pastry, Tapestry and CAN can be
used for this architecture. The routing protocol used is explained in section 5.4.4.

5.4.2 Geographical view of the System

A zoomed out picture of a specific geographic location can be seen from figure 5.1. The
peer nodes are connected with each others in a decentralized manner. The clients can
access these peer nodes or the super peer node.

5.4.3 Node Level Design

Each node has an instance of the run-time system. The run-time system is called as
“The ErlSpace Run-Time System”. “ErlSpace” denotes the combination of erlang and
tuple space. It is designed to have separate components since it would be easy to add
or modify components in the future. This design also enables scaling of individual com-
ponents. The specific components are Tuple Management component, Control compo-
ponent, Network management component, Dispatcher component and Authorizer compo-
ponent. Since the super peer network have different capabilities for different nodes, the
design for the nodes should also be different. Hence, figure 5.3 shows the node level
design for a peer node and figure 5.4 shows the design for a super peer node.

5.4.4 Topology of the Network

The overlay network used is Chord. It is one of the overlay networks which is easy to
implement. There is a two level layer for the chord network [38]. One layer is for the
super peers and the other is for the normal peers. Both peer and the super peer nodes
are hashed using an hash function.

\(^{2}\) All peers have same capability, i.e. have same hardware resources such as processing power, bandwidth.

\(^{3}\) The nodes can differ in their hardware capability and can have different functionality.
Figure 5.2: Super Peer Architecture
5.5 Individual Components

This section explains the individual components of the ErlSpace system along with its functionalities.

5.5.1 Interface Layer

The Interface layer acts as a contact medium for any application to communicate with the ErlSpace runtime system. It has a set of external api\(^4\) calls where the clients can use them in order to talk with the system. It also contains internal api calls which are used to communicate with the other components of the system. The external api’s are carefully designed to be more general and simple rather than application specific. They are as follows:

- out(Tuple),
- inp(Template),
- rdp(Template),
- in(Template),
- rd(Template).

The semantics of these primitives are the same as the original Linda primitives. “Tuple” and “Template” mentioned here are variables. Hence they can store values such as \{4, linda, 10.65\}, \{parallel, cloud\} etc. For example

\(^4\)Application Programming Interface
Chapter 5. System Design/Architecture of the Proposed Solution  

5.5. Individual Components

- `out({4,linda,10.65}),`
- `inp({25,30.300,..})` etc

These api’s are implemented in Erlang with the help of function calls. The ErlSpace primitives used in a client application are:

```erlang
out(Tuple) ->
  Pid ! {out, Tuple}.
inp(Template) ->
  Pid ! {inp, Template}.
rdp(Template) ->
  Pid ! {rdp, Template}.
in(Template) ->
  Pid ! {in, Template}.
rd(Template) ->
  Pid ! {rd, Template}.
```

The asynchronous primitives such as “inp”, “rdp” are very easy to implement in Erlang as by default message passing between processes in Erlang is asynchronous. Any application will have to use only these api’s in their modules. For example consider there is client and server process. Let us consider the client uses the “out” api. In the server side, the interface layer of the system will receive as `out, Tuple}`. The other primitives received at the interface layer would be like `{in, Template}`, `{inp, Template}`, `{rd, Template}` and `{rdp, Template}`. The output from the interface layer would be the same. This will then contact the conversion module of the tuple management component using its internal api’s.
5.5.2 Tuple Management Component

The tuple management component handles the core logic of the system. It consists of three individual modules. They are 1) Conversion module, 2) Tuple space module and 3) Search module. Figure 5.5 shows the individual modules along with the projection of the tuple space with its contents.

![Tuple Management Component Diagram]

The conversion module will receive as \{out, Tuple\} or \{in, Template\} from the interface layer. The format will be like this \{tag, \{Tuple or Template\}\}. It will then check the style of the Tuple. For example

- \{4, 5, linda\} ..... Style is Integer, Integer, String
- \{2, 10\} ..... Style is Integer, Integer
- \{network, cloud, 2, 3.4\} ..... Style is String, String, Integer, Float

For every style a separate record will be created. There can be many records for a particular style. This ensures grouping of tuples inside the tuple space. Depending upon the tag, the conversion module will perform two functions. If the tag is “out” it will then communicate with the tuple space module. Hence the input given to the tuple space module from the conversion module would be a record. If the tag is “in” or “rd” or “inp” or “rdp” it will then communicate with the search module. The input given to the search module would be \{Template\}. The choice of database is implementation specific.
Hence depending on the database used, only the conversion layer needs to be changed accordingly.

The tuple space module can be implemented using the Mnesia database. The choice of mnesia is quite straightforward as it has the following advantages:

- Suitable for data access across cluster of nodes.
- Fast key-value lookups.
- Provides location transparency since the data is distributed.
- Written in Erlang.
- Provides fault-tolerant feature.
- Persistent update.

The items stored inside the mnesia database are in the format of records and can be queried using query list comprehensions (QLC) in Erlang. These queries are similar to Sql\textsuperscript{5} queries. For example the figure 5.6 shows the storage in mnesia and a regular database are very much similar. It will receive records from the conversion module which will then be stored inside the mnesia database.

\textbf{Figure 5.6:} Storage of data inside Mnesia and Relational database

A snippet of code can be like this. The function “get” uses a qlc which queries the table “1” and retrieves all the elements from that table.

\begin{verbatim}
-record(1,{integer, string, integer}).
-record(2,{integer, integer}).

% To add a record into table "1"
add_record(Int, Str, Int) ->
\end{verbatim}

\footnote{\textsuperscript{5}A popular query language used for Relational database management system.}

\begin{verbatim}

\end{verbatim}
\[ R1 = \#1\{\text{integer}=\text{Int}, \text{string}=	ext{Str}, \text{integer}=	ext{Int}\}, \]
\[ F = \text{fun}() \rightarrow \]
\[ \text{mnesia:write}(R1) \]
\[ \text{end}, \]
\[ \text{mnesia:transaction}(F). \]

\% To retrieve all records from table "1"
\$test(select_1) \rightarrow$
\[ \text{get}(\text{qlc:q}([X|\mid X <- \text{mnesia:table}(1)])); \]

The **search module** is used to implement the algorithm for the remote search. The algorithm used here is the same which was used in Tupleware run-time system. It was chosen because, the algorithm used in their system was simple, understandable and easy to implement. The choice of the algorithm totally depends upon the developer of the run-time system. Hence any change or modification will occur only in this search module. It will receive in this format \{Template\}. It will then contact the control component of the system.

### 5.5.3 Control Component

The control component act as an independent body in the run-time system. The control component comes into play during peer registration and search for a remote tuple. As the name suggests, its main functions are controlling the run-time system from getting overloaded and with-stand failures. It consists of two individual modules. They are 1) Registry module and 2) Node maintainence module. Figure 5.7 shows the individual modules. But the functionalities of these modules are slightly different between peer nodes and super peer nodes.

![Control Component](image)

Figure 5.7: Control Component

The **registry module** of the super peer node is used to maintain a list of all the registered peer nodes. It can even be stored in a separate database. If a new peer joins a geographical location, it will then connect to its nearest geographical super peer. This can be compared to the conventional connection as a client node registering with a server node. So this ensures that all the peer nodes are known to the corresponding super peer node. Usually in a regular super peer network design, all the peers would be directly connected to the super peer node. In this super peer network, a peer can be connected to any number of peer nodes and it is enough if any one among them is connected to the super peer network. The figure 5.8 shows the difference. Joining and dis-joining of nodes are controlled by this module.
Chapter 5. System Design/Architecture of the Proposed Solution

5.5. Individual Components

The node maintenance module of the super peer node deals with the node failures, node information and load balancing. Super peer nodes are implemented as supervisors in Erlang. Supervisors look after their worker nodes (peers) and restarts them in case of failures. Since we use mnesia database, even in case of node failures, the data present in the database will be persistent. A problem arises where each super peer network can be a central point of failure. For example, let say, 1000 peers are connected to a super peer node. If the super peer fails, the entire network will fail. Hence all the nodes has to be restarted and there will be lots of communications. This problem can be solved in many ways.

First by having a global supervisor for all the super peer nodes. So if a super peer node fails, the global supervisor will restart it. Manual inspection can be done to maintain the global supervisor.

Second by having a equal super peer node for every geographical location. The two super peers will be connected with each other. So if one fails, the other is used to control the peers. The problem in this approach is the peers would be connected to both the super peers there by increasing the number of network connections.

Third by having multi-levels of supervision as supervision tree. The figure 5.9 shows the supervision tree. Through this approach the supervision is controlled in levels there by reducing the number of communications.

In the peer node, the functionality is different. It maintains the list of neighbours to which it is connected. This list is frequently updated corresponding to the node satisfaction of the tuple request. Due to this a node can be aware of those nodes who satisfy the tuple request and those who does not satisfy. If a node is over loaded of requests or mnesia database is full, the super peer can re-direct the future tuple storage or search request to other peer nodes.

5.5.4 Network Management Component

The network management component acts as a physical layer in the node design. This component forms the main communication link between two physical nodes. Once a tuple comes out of this component, it is open to the network world and it is mostly vulnerable to attacks. Hence security has to be enabled within this component. It consists of two individual modules. They are 1) Encryption module and 2) TCP module\(^6\). Figure 5.10 shows the individual modules.

\(^6\)Transmission Control Protocol. This protocol is used for sending packets across the network.
The function of the Encryption module is used to encrypt and decrypt the template using the crypto module in Erlang. When the relevant tuple for the template is not present in the tuple space then the search module will communicate with the encryption module. The input given to the encryption module would be in this format \{sender_ip_address, op_code, \{template\}\}. This is known as the message structure which would be in the encrypted format and would pass to the TCP module. For example \{192.168.0.1, IN, \{5,8,7,peer\}\}. The parameter op_code contains anyone among the group [RD, RDP, IN, INP].

The TCP module is used to get the encrypted message from the crypto module and then transfer the message across the network. Since the nodes follow a chord layout, the requesting peer node would first contact its corresponding super peer node. This would then forward the request to the relevant super peer. In the other side, this module is used to receive the incoming message and send it to the encryption module which will now decrypt the message.

5.6 Communication Diagrams

This section contains some of the communication pattern diagrams.
### 5.6.1 Simple communication Flow

The figure 5.11 will explain a simple communication flow between the two nodes. The Client “A” is sending a request for a tuple to the node to which it is connected. The nodes then internally do the search for the remote tuple. The modules which are involved in this communication are shown in the diagram. Once the tuple is found, the response is given back to the client.

![Simple Communication Flow Diagram](image)

**Figure 5.11: Simple Communication Flow**

### 5.6.2 Client’s view

The figure 5.12 gives the Client’s view of the entire ErlSpace run-time system. As described in the previous chapter, the client uses only those api’s to communicate with the system. Once it has sent the request for tuple or storage of tuple using the respective api’s and it wait for the responses. The client does not know where the request is being processed. In this way everything is getting abstracted from the client. The abstraction is shown in the diagram. So by this a developer can create small applications which can run from the client side and the rest would be taken care by the run-time system. This ensures the generality feature provided by the ErlSpace system.
5.6.3 Tuple Storage

The figure 5.13 show how a tuple is stored locally. The modules involved in this communication is show in the digram.

5.7 Satisfaction Table for ErlSpace

The criteria’s mentioned in section 3.2 are re-visited again and it justifies how the ErlSpace run-time system would satisfy them theoretically.

The architecture of the ErlSpace system is based on super peer network. A peer-to-peer system in general provides scalability and a decentralized system. In addition to these features, a super peer architecture also handles the hardware resources of the nodes. The super peer nodes acts as a servant, so it can have more hardware capability than
the normal peer nodes. Even when the super peer node get overloaded or failed due to enormous number of peers in each region, this problem can be solved by following any one of the three ways given in section 5.5.3. Hence the system get load balanced. Moreover this architecture have both the advantage of centralized and distributed system. The super peers knows all the peer’s details in their respective regions. Hence they have a faster lookup for data similar to any centralized system. Also they are connected to other super peers through mesh topology which forms a distributed system. From the above we can infer that the ErlSpace system satisfies load balancing and decentralized system criteria’s.

Fault-tolerency is handled by using supervisors in Erlang. If a peer node fails due to network or system crash it would be automatically re-started using supervisors. Suggestions are given when the super peer node fails. So there-by fault-tolerancy and system failure criteria’s is satisfied in the system. Any number of peers can join and leave the system. The super peer or other peer node will get the information of the new peers list. This joining/dis-joining of peers will not affect the working of the system due to the search algorithm used. Since the search algorithm takes decisions based on the local neighbourhood connections rather than on global level. It should be similar to the algorithms used in SwarmLinda and Tupleware systems. The super peer will keep on updating its peer list based on tuple-request satisfied by its peers. Hence the system satisfies the scalability criteria.

Security and reliability is provided by the network management component of the system since the templates are encrypted and passed through a TCP medium. The tuples are distributed across the peer nodes using the Chord layout and decentralized search is provided by the search algorithm used in the system. The grouping of same style of tuples and location transparency is provided since the system uses mneisa database which in-turn is reliable and fault-tolerant. The data would be persistent in case of system failures. Hence, the ErlSpace system satisfies security, distribution of tuples, decentralized search, grouping of tuples and location transparency criteria’s. Finally the api’s provided by the ErlSpace system used by a client application are the same as Linda primitives. There is also provision for defining bulk primitives in the system.

From the above points, it can be seen that the system provides better efficiency (theoretically) due to its distinct features such as super peer model, Erlang. The ErlSpace system satisfies 13/14 criteria’s which is theoretically good when compared with the other systems described in chapter 3.

5.8 Summary

This chapter would have explained the new run-time system for the reader. It would also have given the idea about the system design, architecture and the functionalities of the individual components. The next chapter would tell us the future works which can be carried out based on this architectural design.
Chapter 6

Future Work

This chapter explains the possible future works based on the design and architecture of the proposed solution.

In this thesis work only the design and architecture has been proposed and hence there are lot of things to be developed for the future. Until now it is just a novel proposal, the entire ErlSpace run-time system has to be implemented in order to bring into the reality world. The System can be developed through a phase by phase approach:

1. Phase 1: Make the system to work in one node.
   (a) Implement the external api's such as OUT, RD, IN, RDP and INP.
   (b) Define the internal api's so that the components will communicate with each other.
   (c) Implement the Interface, Conversion, Tuplespace, Search and Registry modules.
   (d) Each node is considered as a server which can be implemented using gen_server module in Erlang.
   (e) Make the entire run-time system to run on a single node.
   (f) Develop a simple client application and try communicating with the system using the implemented api's.

2. Phase 2: Extend this system to support the Super Peer Architecture.
   (a) Define and implement api's for the chord layout.
   (b) Implement the Encryption and TCP modules.
   (c) If necessary implement the Authorizer and Dispatcher component for the super peers.
   (d) Implement supervisor or supervision trees.
   (e) Run the system across a firewall.
   (f) Test the entire system by making a client to request for a remote tuple.

3. Phase 3: Full ErlSpace System is developed.
   (a) Run the system across geographical locations.
   (b) Write test cases for the individual modules.
   (c) Perform integration and performance testing of the entire system.
   (d) Develop other client applications and run on the system.
Chapter 7

Conclusion

As per the requirements of the thesis specification the work has been completed. Initially according to the specification, a prototype has to be developed and verified. But the scope of my thesis was quite big and it was found only after the analysis stage. So we have limited the scope by having a new proposal designed.

The reader would now have gained the knowledge of tuples, templates and the tuple space. The problems in distributing tuple space and its solutions have been analysed and a analytical report was given. From that report, few candidates were selected for possible extension. But none of them were able to be extended due to their individual constraints. Hence ErlSpace system was designed. The system is in its early stages. In future this can be developed in the form of another thesis proposal. Right now, the system proves from its design that it would solve most of problems since it ranks top in the satisfaction table.

I would like to conclude my thesis by saying few things about the experience I had got from Ericsson. The work environment in Ericsson was good and I throughly enjoyed their company. I learned lot of new things about Research & Development and have acquired the skills that are essential for a researcher. This opportunity also helped me, to attend interviews and search for jobs. Finally I would like to say that it was an overall good learning experience.
Bibliography


[13] Song Qin, *Merge Sort Algorithm*, Dept. of Computer Sciences, Florida Institute of Technology, Melbourne, FL 32901. 11


Appendix A

Merge Sort Application

A.1 Installation Manual

This section will provide the steps to install the Tupleware system and run the merge sort application on top of it.

1. Download the Tupleware source code package from the GitHub repository\(^1\).

2. The software which can be used are Java SDK version 7 or later, Apache Ant for compilation and Linux operating system. Or you can use Eclipse integrated development environment version 3.5 or later.

3. Extract the contents and it will be stored in the folder called “tupleware”.

4. Open the contents using eclipse and the project name would be “scope”.

5. Open a new java project and name it as mergesort and copy the source code below into the respective files as “mergesort.java”, “mergesortMaster.java” and “mergesortWorker.java”.

6. Make these changes in the “tuplespaceruntime.java” file
   - Give the number of worker nodes in the NODES field.
   - Give the relevant ipaddress and port number in the GTS_ADDRESS field.
   - In the “getlocalIPAddress()” function call, start the variable “i” from “0”.

7. Open the “properties dialog” for the mergesort project. Link the project “scope” to the Projects tab inside Java Build Path section.

8. In the same section, check whether the project “scope” has been added and clicked in the Order and Export tab. Finally click OK.

9. Compile the files using the built-in ant compiler.

10. Give the initial parameters such as port number, number of elements and threshold value in the Run Configurations dialog box for both mergesortMaster and mergesortWorker files.

11. Finally the run the mergesortMaster and depending upon the number of worker nodes, run the mergesortWorker.

\(^1\) [https://github.com/akatkinson/Tupleware](https://github.com/akatkinson/Tupleware) as on December 19, 2011.
A.2 Source Code

% Merge Sort Master
import java.util.*; //java.util.Vector
import runtime.*;
import space.*;

public class mergesortMaster {
    private TupleSpaceRuntime ts;
    private int port1;
    private int no_of_elements;
    private int threshold;
    public final int no_of_parts = 5;
    public mergesortMaster(){}
    public mergesortMaster(int port, int elements, int threshold){
        this.no_of_elements = elements;
        this.port1 = port;
        this.threshold = threshold;
        //System.out.println("Number of Elements to be Sorted is " + no_of_elements);
        ts = new TupleSpaceRuntime(port1,true);
    }
    public void start(){
        System.out.println("Master process started..........");
        Vector<Integer> a = new Vector<Integer>(no_of_elements);
        mergesort m = new mergesort();
        System.out.println("Initialization of the Array values....");
        m.initdata(a, no_of_elements);
        System.out.println("Done");
        System.out.println("-----------Initializing the Tuppleware Runtime System--------");
        ts.start();
        System.out.println("-----------Initializing Done--------");

        /* Divides the Array into Partition and randomly send the Partition to the worker nodes */
        //System.out.println("\n Dividing the Array into Partition... and Sending them to Random Worker Node");
        Vector<List<Integer>> parts = divide_array(a);
        System.out.println("NO OF PARTITIONS ARE: " + parts.size());
        for(List<Integer> part: parts){
            mergesort ms = new mergesort(part,threshold);
            ts.outRand(new Tuple("mergesort",ms,"unsorted"));
        }
        System.out.println("Done.\n\n");

        /* Collecting the Sorted partitions */
    }
}
int n = 0;
Vector<List<Integer>> sortedPartitions =
new Vector<List<Integer>>(0);
System.out.print("Waiting for data to be sorted...");
while(n < no_of_elements){
TupleTemplate template =
new TupleTemplate("mergesort",null,"sorted");
Tuple t = ts.ts.in(template); //keeps on retrieving from its
local tuple space
mergesort ms = (mergesort) t.field(1);
List<Integer> v = ms.get_data();
sortedPartitions.addElement(v);
n = n + v.size();
System.out.println("collected " + n + " elements.");
}
System.out.println("Done. " + sortedPartitions.size() + " partitions collected.");

/* Distribute poison pill to all the workers */
ts.outEach(new Tuple("mergesort", new mergesort(),"complete");

/* Reconstructing the array */
List<Integer> left = new ArrayList<Integer>();
List<Integer> right = new ArrayList<Integer>();
System.out.println("Master is Rconstructing the Array....Please Wait...");
while (!(sortedPartitions.size() == 1)){
mergesort o = new mergesort();
List<Integer> temp = new ArrayList<Integer>();
left = sortedPartitions.get(0);
right = sortedPartitions.get(1);
sortedPartitions.removeElementAt(0);
sortedPartitions.removeElementAt(0);
temp = o.merge(left, right);
sortedPartitions.add(0, temp);
}
System.out.println("Array Elements are:" + sortedPartitions.elementAt(0).toString());
System.out.println("Sorting is Completed:

// Stopping the Runtime System
System.out.println("Stopping the Runtime System");
ts.stop();
System.out.println("Done");
}
public Vector<List<Integer>> divide_array(Vector<Integer> array){
Vector<List<Integer>> parts = new Vector<List<Integer>>(0);
List<Integer> temp = null;
int part_size = array.size()/no_of_parts;
int fromIndex = 0;
int toIndex = 0;
int lastelementIndex = array.lastIndexOf(array.lastElement());

for(int j = 0; j <= lastelementIndex; j = j + part_size){
fromIndex = j;
toIndex = toIndex + part_size;
List<Integer> sublist = null;
if (toIndex < array.size()){  
temp = array.subList(fromIndex,toIndex);
sublist = new ArrayList<Integer>(temp);
parts.add(sublist);
}
else{
toIndex = array.lastIndexOf(array.lastElement());
if (fromIndex != toIndex){
toIndex = toIndex + 1;
temp = array.subList(fromIndex,toIndex);
sublist = new ArrayList<Integer>(temp);
parts.add(sublist);
}
else{
temp = array.subList(0,1);
sublist = new ArrayList<Integer>(temp);
parts.add(sublist);
}
}
return parts;
}

public static void main(String[] args) {
if(args.length != 3) {
    System.out.println("Usage: <PORT> <VALUES> <THRESHOLD> ");
    System.exit(1);
}

    int port   = Integer.parseInt(args[0]);
    int elements = Integer.parseInt(args[1]);
    int threshold = Integer.parseInt(args[2]);

    mergesortMaster master = new mergesortMaster(port,
    elements,threshold); //new operator is followed by a constructor
    master.start();
}

%Merge Sort Worker
import space.*;
import runtime.*;
import java.io.*;
import java.util.*;

public class mergesortWorker {

private int port1;
private TupleSpaceRuntime ts;

public mergesortWorker(int port){
this.port1 = port;
ts = new TupleSpaceRuntime(port1,false);
}

public void start(){

System.out.println("--------Initializing the
Tupleware Runtime System--------\n");
ts.start();
System.out.println("----------Initializing Done--------");

System.out.println("Worker process started.");

System.out.println("Attempting to fetch the Partitions...");
TupleTemplate template = new TupleTemplate("mergesort",null,"unsorted");
Tuple t = ts.ts.in(template); // it checks in its local tuple space
//Tuple t = ts.in(template); // it checks in its local tuple space then remotespaces
System.out.println("Fetching...Done.");

mergesort ms = (mergesort) t.field(1);
String status = (String) t.field(2);

while(!status.equals("complete") ) {
// sort if it is less than threshold OR split it again
boolean check = ms.ready();
if(check){
System.out.println("Sorting " + ms.size() + " items.");
List<Integer> sorted = ms.merge_sort(ms.get_data());
//System.out.println("Sorted (partial)
elements are" + sorted + "and size is " + sorted.size());
mergesort msort = new mergesort(sorted);
try{
ts.gts.out(new Tuple("mergesort",
msort, "sorted")); // stores back to the master tuple space
}catch(IOException e) {
System.out.println("Error when returning
sorted partition to master process");
}

}
A.2. Source Code

Appendix A. Merge Sort Application

```java
System.out.print("Successfully sorted partition.\n\nAttempting to fetch more data...");
t = ts.in(new TupleTemplate("mergesort", null, null)); // first checks in its local tuple space
    // then searches the remote spaces
System.out.println("Done.");
ms = (mergesort) t.field(1);
status = (String) t.field(2);
}
else{
    System.out.print("Splitting the\n(Partition)array of size " + ms.size());
    mergesort dup = ms.split(ms.get_data());
    System.out.println(" into " + ms.size() + "/" + dup.size());
}
System.out.print("Split partition..continuing...");
//ts.out(new Tuple("mergesort", dup, "unsorted")); // stored in the worker’s local tuple space
ts.outRand(new Tuple("mergesort", dup, "unsorted")); // stored randomly in worker’s remote space
}
System.out.print(".");
}
System.out.println();
System.out.print("Worker process finished...Shutting down...");
ts.out(new Tuple("mergesort", new mergesort(), "complete")); // try to get all processes to shut down

// Stopping the Runtime System
    System.out.println("Stopping the Runtime System");
    ts.stop();
    System.out.println("Done.");
}

public static void main(String[] args) {
    if(args.length != 1) {
        System.out.println("Usage: <PORT>");
        System.exit(1);
    }
    int port = Integer.parseInt(args[0]);
    mergesortWorker worker = new mergesortWorker(port);
    worker.start();
}

% Merge Sort
```
import java.util.*;

public class mergesort implements java.io.Serializable{

    private static final long serialVersionUID = 1L;
    private List<Integer> array;
    private int threshold;

    public mergesort(){
    }

    public mergesort(List<Integer> array, int threshold){
        this.array = array;
        this.threshold = threshold;
    }

    public mergesort(List<Integer> array){
        this.array = array;
    }

    /*Dividing the Array until it has one element */
    public List<Integer> merge_sort(List<Integer> a){
        List<Integer> r = new ArrayList<Integer>();
        int total = a.size();
        /*Checking the base condition*/
        if (total <= 1) {
            return a;
        }
        List<Integer> left = new ArrayList<Integer>();
        List<Integer> right = new ArrayList<Integer>();
        int middle = total/2;
        for(int j = 0; j < middle; j++)
            left.add(j, a.get(j));
        for(int i = 0; middle < a.size() ; i++,middle++)
            right.add(i, a.get(middle));
        left = merge_sort(left);
        right = merge_sort(right);
        r = merge(left, right);
        //System.out.println(r.toString());
        return r;
    }

    public List<Integer> merge(List<Integer> left, List<Integer> right){
        List<Integer> result = new ArrayList<Integer>();
        int k = 0;
        }
while (left.size() > 0 || right.size() > 0) {
    if (left.size() > 0 && right.size() > 0) {
        if (left.get(0) <= right.get(0)) {
            result.add(k, left.get(0));
            left.remove(0);
            left = rest(left);
        } else {
            result.add(k, right.get(0));
            right.remove(0);
            right = rest(right);
        }
    } else if (left.size() > 0) {
        result.add(k, left.get(0));
        left.remove(0);
        left = rest(left);
    } else if (right.size() > 0) {
        result.add(k, right.get(0));
        right.remove(0);
        right = rest(right);
    }
    k++;
}
return result;

public List<Integer> rest(List<Integer> list) {
    List<Integer> l = new ArrayList<Integer>();
    List<Integer> sublist = new ArrayList<Integer>();
    int lastIndex = 0;
    if (!list.isEmpty()) {
        if (list.size() != 1) {
            lastIndex = list.lastIndexOf(list.get(list.size() - 1)) + 1;
        } else {
            l = list.subList(0, 1);
        }
    }
    l = list.subList(0, lastIndex);
    sublist = new ArrayList<Integer>(l);
    return sublist;
}

/* Initialization of the data */
public void initdata(Vector<Integer> a, int elements) {


Random rand = new Random(System.currentTimeMillis());
// System.out.println(a.size());
for(int i = 0; i < elements; i++){
a.add(i, rand.nextInt());
}
//System.out.println("The Array elements are: " + a);

public boolean ready() {
    return (array.size() <= threshold);
}

public int size(){
    return array.size();
}

public int firstpartsize(){
    return firstpart.size();
}

public int secondpartsize(){
    return secondpart.size();
}

public List<Integer> get_data() {
    return array;
}

public List<Integer> get_firstpart() {
    return firstpart;
}

public List<Integer> get_secondpart() {
    return secondpart;
}

public mergesort split(List<Integer> part) {
    int j = 0;
    int k = part.size();
    List<Integer> belowThreshold = null;
    List<Integer> aboveThreshold = null;
    if(j < k ){
        int fromIndex = part.indexOf(part.get(0));
        int toIndex = threshold;
        List<Integer> a = part.subList(fromIndex,toIndex);
        belowThreshold = new ArrayList<Integer>(a);
        List<Integer> b = part.subList(threshold, part.size());
        aboveThreshold = new ArrayList<Integer>(b);
    }
}
} else {
    this.array = part;  // CHECK
}
this.array = belowThreshold;
mergesort dup = new mergesort(aboveThreshold, threshold);

    return dup;

}