**FINAL YEAR PROJECT REPORT**

**A SCALABLE AND FAULT-TOLERANT ARCHITECTURE FOR COMPLEX EVENT PROCESSING SYSTEMS**



**Department of Computer Science and Engineering**

**University of Moratuwa**

**Project Group: PI-16**

**Group Members:**

|  |  |
| --- | --- |
| Harshana Eranga Martin | 060295J |
| Dishan Sachindra Metihakwala | 060305T |
| Hettige Chathura Randika | 060398D |
| Rajeev Ranga Sampath | 060428X |

**Supervisors**:

Mr. K. Sarveswaran (Internal)

Mr. Manusha Wijekoon (External)

Dr. Buddhinath Jayathilake (External)

**Coordinator:**

Mr. Shantha Fernando

**Date of Submission: 21st May 2010**

This report is submitted in partial fulfilment of the requirements for the award of the degree of Bachelor of Science in Engineering at the University of Moratuwa, Sri Lanka

# Abstract

Project Title: epZilla (Scalable Fault Tolerant Architecture for Complex Event Processing Systems)

Authors : H.E. Martin, D.S. Methihakwala, D.M.R.R. Sampath, H.C. Randika

Supervisor/s: Mr. K. Sarveswaran (Internal)

Mr. Manusha Wijekoon (External)

Dr. Buddhinath Jayathilake (External)

Project epZilla is an effort to build highly scalable, highly available, fault tolerant distributed system to use in Complex Event Processing systems. The main objective of the project is to build efficient processing platform to deploy Complex Event processing agents. In this Report, we introduce Project epZilla in vast amount of detail. This includes the details on conception and design, its development and implementation as well as the results and outcomes that we have achieved throughout the process. We now present epZilla as a complete solution that can be use in distributed complex event processing systems.

Key words:

epZilla, Distributed Systems, CEP, STM, Dynamic Discovery, Stratification, Java RMI

# Acknowledgement

The authors of this report wish to acknowledge, with greatest honesty, the guidance, support, encouragement and assistance provided by the following persons in various aspects to successfully complete the project.

First and foremost, we would like to thank the coordinator and the supervisors of the project for their continued presence and assistance throughout the project. From the Department of Computer Science, Mr. Shantha Fernando, coordinator of the final year projects and Mr. K. Sarveswaren internal supervisor has always guided us in our final year project. Also it is great pleasure to thank Mr. Manusha Wijekoon from Creative Solutions (Pvt) Ltd, who originally proposed the project idea and gave us continuous support till the end to successfully complete the project.

Also we would really appreciate the advices given by Ms. Vishaka Nanayakkara, Head of the Department of Computer Science and Engineering and Dr. Shahani Weerawarana senior lecturer Department of Computer Science and Engineering. We also like to thank the valuable ideas given by Dr. Buddinath Jayathilaka during our project.

Also academic and non academic staff members at Department of Computer Science and Engineering, gave us support in arranging labs for our testing.

We would also like to appreciate the support, assistance, ideas and opinions by our friends and fellow undergraduates, and, more importantly, for pulling us up when we feel down.

--epZilla team--

# Table of Contents

[Abstract i](#_Toc262417481)

[Acknowledgement ii](#_Toc262417482)

[Table of Contents iii](#_Toc262417483)

[List of Figures vi](#_Toc262417484)

[List of Tables viii](#_Toc262417485)

[List of Symbols and Units ix](#_Toc262417486)

[Chapter 1 1](#_Toc262417487)

[1.1 Introduction 1](#_Toc262417488)

[1.1.1       Complex Event Processing (CEP) 2](#_Toc262417489)

[1.1.2       Distributed Systems 2](#_Toc262417490)

[1.1.3       Fault Tolerance (FT) 3](#_Toc262417491)

[Faults 3](#_Toc262417492)

[Errors 3](#_Toc262417493)

[Failures 3](#_Toc262417494)

[Fault Tolerance Procedure 3](#_Toc262417495)

[1.1.4       Scalability 4](#_Toc262417496)

[1.1.5 Load distribution 5](#_Toc262417497)

[Chapter 2 – Aims and Objectives 6](#_Toc262417498)

[2.1 Introduction 6](#_Toc262417499)

[2.2 Objectives of the Project 6](#_Toc262417500)

[Chapter 3 – Literature Survey 8](#_Toc262417501)

[3.1 Concepts of Complex Event Processing 8](#_Toc262417502)

[3.1.1 Events 8](#_Toc262417503)

[3.1.2. Patterns of Events 8](#_Toc262417504)

[3.1.3 Event pattern constraints 9](#_Toc262417505)

[3.1.4 Event pattern triggered rules 9](#_Toc262417506)

[3.1.5 Complex events 9](#_Toc262417507)

[3.2    Existing Complex Event Processing Solutions 9](#_Toc262417508)

[3.2.1 Oracle Complex Event Processing 10](#_Toc262417509)

[3.2.1 Aleri's Complex Event Processing 10](#_Toc262417510)

[3.3    Concepts of fault tolerance in distributed systems 11](#_Toc262417511)

[3.3.1    Fault classification: 11](#_Toc262417512)

[3.3.2    The following procedures have been used in distributed system design: 12](#_Toc262417513)

[3.3.3 Fault tolerant algorithms 13](#_Toc262417514)

[3.4    Replication techniques 14](#_Toc262417515)

[3.4.1    Active replication 14](#_Toc262417516)

[3.4.2    Passive replication 15](#_Toc262417517)

[3.4.3    Semi active replication 16](#_Toc262417518)

[3.4.4 Problems in replication techniques 16](#_Toc262417519)

[3.5    Software Transactional Memory 16](#_Toc262417520)

[3.6    Check pointing techniques 19](#_Toc262417521)

[3.6.1    Coordinated check-pointing 20](#_Toc262417522)

[3.6.2    Uncoordinated check-pointing 20](#_Toc262417523)

[3.6.3    Disk-less check-pointing 21](#_Toc262417524)

[3.6.4    Communication induced check-pointing 21](#_Toc262417525)

[3.7    Log based rollback recovery 21](#_Toc262417526)

[3.8 Stratification 22](#_Toc262417527)

[3.9 Scalability and load distribution in distributed systems 23](#_Toc262417528)

[3.10 Leader in Distributed Systems 25](#_Toc262417529)

[3.11 Election Algorithms for Distributed Systems 26](#_Toc262417530)

[3.12 Dynamic Service Discovery 27](#_Toc262417531)

[Chapter 4 - Project Design Overview 30](#_Toc262417532)

[4.1 Introduction 30](#_Toc262417533)

[4.2 System Architecture 30](#_Toc262417534)

[4.3 Client side operation 32](#_Toc262417535)

[Chapter 5 - System Implementation 33](#_Toc262417536)

[5.1 epZilla Client 33](#_Toc262417537)

[5.2 Name Server 33](#_Toc262417538)

[5.3 Dispatcher 35](#_Toc262417539)

[5.4 Cluster Node 36](#_Toc262417540)

[5.5 Accumulator 38](#_Toc262417541)

[5.6 Software Transactional Memory 39](#_Toc262417542)

[5.6.2 Dispatcher STM Implementation 39](#_Toc262417543)

[5.6.3 Cluster and Accumulator STM Implementation 41](#_Toc262417544)

[5.7 Stratification and trigger distribution 41](#_Toc262417545)

[5.7.1 Implementation details of dynamic dependency analysis 42](#_Toc262417546)

[5.7.2 Trigger base re-organization 44](#_Toc262417547)

[5.7.3 Implementation of trigger re-organization 45](#_Toc262417548)

[5.8 Dynamic Service Discovery 46](#_Toc262417549)

[5.9 Leader Election 55](#_Toc262417550)

[5.10 Load Balancing 65](#_Toc262417551)

[5.11 Remote Method Invocation 66](#_Toc262417552)

[5.12 Log based recovery 67](#_Toc262417553)

[Chapter 6 – Tests, Results and Analysis 69](#_Toc262417554)

[6.1 Introduction 69](#_Toc262417555)

[6.2 Unit Tests and Results 69](#_Toc262417556)

[6.3 Integration Tests and Results 71](#_Toc262417557)

[6.4 Performance Tests and Results 72](#_Toc262417558)

[6.4.1 XSTM performance results 72](#_Toc262417559)

[6.4.2 Dispatcher Event Distribution results 75](#_Toc262417560)

[6.4.3 Dynamic trigger dependency analysis 79](#_Toc262417561)

[Chapter 7 – Discussion 80](#_Toc262417562)

[7.1 Introduction 80](#_Toc262417563)

[7.2 The Outcome 80](#_Toc262417564)

[7.3 Our achievements 81](#_Toc262417565)

[Chapter 8 – Future Work 83](#_Toc262417566)

[8.1 Improvements to the STM 83](#_Toc262417567)

[8.2 Improvements to the Dynamic Discovery Mechanism 83](#_Toc262417568)

[8.3 Improvements to the Dynamic Load Balancing 84](#_Toc262417569)

[8.4 Improvements to System Initialization 84](#_Toc262417570)

[Chapter 9 – Conclusion 85](#_Toc262417571)

[References 86](#_Toc262417572)

# List of Figures

[Figure 1: Active Replication 15](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417573)

[Figure 2: Passive Replication 15](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417574)

[Figure 3: The Structure of XSTM 18](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417575)

[Figure 4: System Architecture 31](#_Toc262417576)

[Figure 5: Event flow of the System 32](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417577)

[Figure 6: IP Comparator algorithm 34](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417578)

[Figure 7: Dispatcher overview 35](#_Toc262417579)

[Figure 8: Node component 37](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417580)

[Figure 9: Accumulator Overview 38](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417581)

[Figure 10: Add list of triggers to STM 39](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417582)

[Figure 11: Add triggers to shared list 40](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417583)

[Figure 12: Dependency Graph 44](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417584)

[Figure 13: Jini Discovery Mechanism 48](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417585)

[Figure 14: epZilla Dynamic Service Discovery Mechanism 49](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417586)

[Figure 15: Abstract View of the Dynamic Discovery API 50](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417587)

[Figure 16: Code Segment Interface IService Publisher 50](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417588)

[Figure 17: Code Segment for DispatcherPubliser Implementation 51](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417589)

[Figure 18: Code Segment DispatcherPubliser (Cont.) 52](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417590)

[Figure 19: Code segment for publisherService implmentation. 52](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417591)

[Figure 20: Code segment for addSubscription implementation 53](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417592)

[Figure 21: Dispatcher Dynamic Discovery mechanism overview 53](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417593)

[Figure 22: Code segment for MulticastMessageDecoder 54](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417594)

[Figure 23: Code segment for TCPMessageDecoder 54](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417595)

[Figure 24: Code segment for TCPMessageDecoder (Cont.) 55](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417596)

[Figure 25: Formal Definition of LCR Algorithm[20] 57](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417597)

[Figure 26: Code segment of LCR Implementation 57](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417598)

[Figure 27: Code segment of Status enumerator class in LCR Implementation 57](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417599)

[Figure 28: Code Segment of EpzillaIpConfig configuration file 58](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417600)

[Figure 29: Cluster ID definition in the configurarion file. 58](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417601)

[Figure 30: Code Segment of Message Id definition. 59](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417602)

[Figure 31: Java RMI message protocol 59](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417603)

[Figure 32: Code segment for RMI Message Decoder 60](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417604)

[Figure 33: Event Driven Architecture Implementation 61](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417605)

[Figure 34: Code Segment of LCR Algorithm Logic Implementation 62](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417606)

[Figure 35: State Transitions and message passing after executing the LCR Logic. 62](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417607)

[Figure 36: State Transitions and message passing after executing the LCR Logic (Cont.) 63](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417608)

[Figure 37: Leader Election initiation process 63](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417609)

[Figure 38: Leader Election Initiation process 64](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417610)

[Figure 39: Default Leader Definition 64](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417611)

[Figure 40 : RMI Messaging Overview 66](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417612)

[Figure 41: Log Based Recovery algorithm 67](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417613)

[Figure 42: Checkpoint data format 68](file:///C:\Documents%20and%20Settings\Administrator\My%20Documents\My%20lec%20present\Level%204\CS%204200%20-%20Research%20Project\Final%20Report\fypreport\Draft%20Report%20-%20PI%2016-%20project%20epZilla.docx#_Toc262417614)

[Figure 43: Transaction rate Vs Object size graph 73](#_Toc262417615)

[Figure 44: Network usage Vs Object size graph 74](#_Toc262417616)

[Figure 45: Event Dispatching (Simultaneously with Triggers) 77](#_Toc262417617)

[Figure 46: Event Dispatch Rates 78](#_Toc262417618)

# List of Tables

[Table 1: Multicast Message Test 69](#_Toc262417619)

[Table 2: Leader Election verification Test 70](#_Toc262417620)

[Table 3: Trigger Distribution Test 71](#_Toc262417621)

[Table 4: Event Distribution Test 72](#_Toc262417622)

[Table 5: Transation Rates Vs Object size 73](#_Toc262417623)

[Table 6: Network Usage Vs Object Size 74](#_Toc262417624)

[Table 7: The Tesing Environment 75](#_Toc262417625)

[Table 8: Event Dispatch rates for simultaneous process 76](#_Toc262417626)

[Table 9: Event Dispatch rates 78](#_Toc262417627)

# List of Symbols and Units

API Application Programming Interface

CEP Complex Event Processing

DD Dynamic Discovery

EDA Event Driven Architecture

GUI Graphical User Interface

IDE Integrated Development Environment

IO Input/output

IP Internet Protocol

IT Information Technology

LAN Local Area Network

LCR Le Lann, Chang & Roberts

LE Leader Election

OS Operating System

OSGi The OSGi Alliance (formerly known as Open Services Gateway Initiative)

R&D Research and Development

RMI Remote Method Invocation

RS Remote Services

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

STM Software Transactional Memory

TCP Transmission Control Protocol

UI User Interface

UML Unified Modeling Language

UOM University of Moratuwa

WS Web Services

# Chapter 1

## Introduction

Complex event processing which is also called CEP is one of the emerging areas in Information and Communication Technology. Complex Event Processing is developed based on the concepts of Event Processing.

Complex event processing involves searching an event stream against a set of queries known as triggers to identify meaningful patterns declared by the set of triggers. Identifying these patterns is important in many applications such as financial market data analysis, air traffic scheduling, etc.

Fault tolerance is a very important property in terms software engineering and it means the capability of a system to perform its functions in the desired manner even in the presence of erroneous situations.

Project epZilla is focusing on implementing a Scalable and Fault Tolerance Architecture for Complex Event Processing Systems. In this project two main technological areas are involved. They are Complex Event Processing and Distributed Systems. Since Complex Event Processing is heavily processing oriented, Distributed Systems provide an efficient processing platform for the Complex Event Processing Agents. This project is mainly focusing on implementing a Distributed Architecture with Scalability and Fault Tolerance capabilities for CEP rather than implementation of Complex Event Processing.

Project epZilla is a highly research based project which involves a considerable amount of implementation work as well. This idea was proposed by an employee from the industry, Mr. Manusha Wijekoon from Creative Solutions (PVT) LTD. The initial project idea is to implement a Scalable and Fault Tolerance Distributed Architecture for Complex Event Processing systems.

### 1.1.1       Complex Event Processing (CEP)

Complex event processing means, searching a given event stream against a given set of queries and revealing the meaningful relationships among these events based on the given set of queries. CEP systems are capable of conducting operations such as complex pattern matching, event correlation, abstraction, hierarchical organization, causality, membership, timing etc [1]. In this context we can consider a set of transactions in a stock market as the stream of events and we can consider a set of rules like Time difference between transactions of the same person for the same symbol, no of consecutive stock selling transactions for the same symbol, no of consecutive stock buying transactions for the same symbol, etc.  In complex event processing, the system searches for any matching set of transactions, in the given set of transactions. If there is any matching set of transactions, system will send a reply as an alert message or by some other means.

Complex event processing systems are involved in many territories such as,

* Search Documents for a set of specific key words
* Radio Frequency Identification analysis
* Financial market transaction pattern analysis
* Banking Applications
* Air traffic scheduling

### 1.1.2       Distributed Systems

A Distributed System is a system that uses multiple hardware units, typically hosts that are used collectively to provide a service [2]. These hardware units can be hosts, networks, routers, switches, etc. In a typical enterprise level distributed system, these hardware units are connected using a high speed network for intercommunication. Functionality of the system is distributed among many processes. These processes can run on many machines, depending on their requirements on processing power, memory and input-output. Distributed systems are a form of parallel processing. Multicomputer Distributed systems allow us to create redundant processing units with same computer programs running in different hosts. So each machine can be used to process same input without any difficulty. In this scenario, distributed systems can work as a mirror processing unit for the original processing unit. Hence such a distributed system can be used for load balancing. This complete distributed system can have a high throughput and low latency in processing an input queue.

### 1.1.3       Fault Tolerance (FT)

Fault tolerance is the capability of a system to perform its functions in the desired manner even with the presence of erroneous situations [3]. Usually FT is achieved through redundancy. This implies if a component fails, one or more of the non-failed replicas will continue to provide service with no appreciable disruption.

### Faults

Faults are the root causes of errors. When faults are present errors may occur. But not every fault causes an error. It is very difficult to track faults unless errors are detected.

### Errors

Errors are the symptoms of faults and they represent invalid states in the system and lead to failures.

### Failures

When a system deviates from its actual behaviour, we call it, a failure. The cause of failures is errors.

### Fault Tolerance Procedure

FT procedure comprises 4 main stages, namely Error detection, Damage confinement, Error recovery, Fault treatment. [3]

#### Error Detection

Error detection is the first step of FT procedure. In this stage, errors are identified using various techniques such as Replication checks, Timing checks, Run-time constraints checking, Diagnostic checks, etc.

#### Damage Confinement

At this stage, each and every component along the information and data flow, starting from known failed components are checked for errors and the boundaries of the error are identified. Then the identified components are isolated to fix the faults.

#### Error Recovery

This stage focuses on recovering from the error. There are 2 well known mechanisms of recovering errors. They are Backward error recovery mechanisms such as Check pointing, Replication and Forward error recovery mechanisms.

#### Fault Treatment

In this phase, the actual fault is being treated and fixed. Usually what happens is, the faulty component is implemented again and integrated to the system.

### 1.1.4       Scalability

Scalability is the ability of the system to handle increasing loads while maintaining other performance criteria such as throughput, latency, etc [4]. Scalability is very important in enterprise scale applications as it allows smooth operation of the system throughout its lifetime, without requiring large upgrades when load is increased. Hence scalability is a very important factor in software development.

In this project we consider scalability in terms of events and triggers. The system must be capable of handling increment in event stream and increment in trigger base.

Scalability also can be achieved using redundancy and extensions. As an example, if we integrate a backup server which actively involves in request processing as a Fault tolerance mechanism, then the systems capability to handle additional requests increases, thus the scalability in terms of requests is increased in the system.

### 1.1.5 Load distribution

Load distribution is the process of transferring units of work, among processing elements during execution, to maintain balance of the processing elements.

# Chapter 2 – Aims and Objectives

## 2.1 Introduction

Objective of this chapter is to make reader understand about the aims and major objectives of the project epZilla. Following sub section described about the major aims, objectives of the project and the final deliverables of the project.

## 2.2 Objectives of the Project

The Primary goal of the project epZilla is to create a Scalable fault tolerant architecture for complex event processing systems. In successfully obtaining the desired goal, the project tackles a vast number of trivial, yet daunting problems, which are common in the design of any reliable distributed system, plus another set of specific requirements, generated by the concepts of complex event processing. The main objectives of the project are as follows.

* The design and implementation of a Scalable fault tolerant architecture for complex event processing systems.

The proposed architecture would handle scalability and fault tolerance in the architecture itself without imposing any extra requirements to the system itself.

* Designing the architecture such that it provides complex event processing support up to any number of complexity levels which would depend on the usage of the system.

Complex event processing is defined as having multiple complexity levels. Even though most modern usages of complex event processing require only a few complexity levels, the architecture would provide the user with the flexibility of implementing any number of complex event processing levels, as requested by their application.

* Designing the architecture such that it requires very little or no hardware modifications in a client’s generic distributed system.

The architecture provides a comprehensive software solution which requires no modifications to the distributed system hardware. Eg- number of network interfaces per node, the number of LAN switches, the orientation of the network links etc.

* Designing the architecture such that it provides a practically unlimited scope of scalability for the usage in any general complex event processing scenario.

The architecture will have practically unlimited scalability in the scopes of both the number of events processed per unit time and the number of triggers that have to be evaluated by the system.

* Designing the architecture such that it uses the available resources in a practically optimal manner to achieve the maximum possible efficiency while processing events.

The proposed architecture aims to have a minimum number of non event processing support nodes to gain an optimal level of efficiency and to restrict resource wastage.

# Chapter 3 – Literature Survey

# 3.1 Concepts of Complex Event Processing

Complex Event Processing, best known as CEP is a technology for low-latency filtering, correlating, aggregating, and computing on real- world event data.  It primarily tries to identify specific patterns through a stream of events. CEP analyses the stream of event data in real-time to generate immediate insight and enable instant response to changing conditions. CEP can provide an organization with the capability to define, manage and predict events, situations, exceptional conditions, opportunities and threats in complex, heterogeneous networks. These observations help to improve the operational situational awareness in many business scenarios [5][6].

### 3.1.1 Events

An event is an object that represents or records an activity that happens, or is thought of as happening. Examples are:

• A purchase order (records a purchase activity).

• An email confirmation of an airline reservation.

• Stock tick message that reports a stock trade.

• A message that reports an RFID sensor reading.

### 3.1.2. Patterns of Events

In the most general form of CEP, an event pattern can be a timing or causal relationship between patterns of events as well as a Boolean combination. A pattern contains variables which are replaced by values to form instances of the pattern. A pattern can have many different instances [7].

### 3.1.3 Event pattern constraints

An event pattern constraint is a pattern that should never occur in the activity of the enterprise. A constraint is a negation of an event pattern.

### 3.1.4 Event pattern triggered rules

In CEP event patterns are used to trigger reactive rules and take actions when those patterns occur.

### 3.1.5 Complex events

An event that is an abstraction of other events called its members.

Examples:

• The 1929 stock market crash (an abstraction denoting many thousands of member events, including individual stock trades),

• The 2004 Tsunami (an abstraction of many natural events)

• A CPU instruction (an abstraction of register transfer level (RTL) events),

• A completed stock purchase (an abstraction of the events in a transaction to purchase the stock).

## 3.2    Existing Complex Event Processing Solutions

There are number of existing commercial and open source complex event processing solutions in the industry today. But none of them utilize a scalable and reliable distributed system to conduct their actual event processing. Most of the solutions depend on powerful and expensive servers to conduct their event processing and most of the time the complete procedure of event processing is done on a single machine. Most of the solutions were aimed at very large organisations which were prepared to spend a vast amount of resources to obtain a CEP engine.

The approach taken towards complex event processing from this project severely differs from the existing solutions in several aspects. The proposed architecture is a distributed system which is scalable enough to fit the requirements of organisations of any scale and capability. The distributed system can be constructed from relatively inexpensive computers. Since the cost of processing power has reduced hugely in the past decade, organisations can construct the architecture at a required scale depending on their budget and requirements relatively easily. The move to a distributed system brings forward the concept of failures with it, but the features of fault tolerance built in to the architecture would make it extremely reliable. Again the architecture is designed to carter for the amount of reliability needed by the organisation in an extremely flexible manner.

A few of the existing complex event processing solutions would be briefly explained in the following sections

### 3.2.1 Oracle Complex Event Processing

The Oracle Complex Event Processor provides a declarative environment for the development of event processing applications that can process and act on hundreds of thousands of events per second. The system is capable of handling key CEP features which include pattern matching, user-defined windows for event evaluation and the contextual enrichment of events. Oracle uses an extension to SQL called Oracle Continuous Query Language or CQL, to enable development of CEP based applications. Oracle CEP can be deployed as a stand-alone application on third party application servers or as an integrated service engine within the Service Infrastructure of the Oracle Application Server 11g.

### 3.2.1 Aleri's Complex Event Processing

Aleri's Complex Event Processing technology is designed to quickly build and deploy applications that analyze streaming event data in real-time to help their clients for better decision making and immediate response to changing conditions. Aleri uses the Coral8 Engine™ which is a Complex Event Processing platform that delivers the information needed by the business. Aleri provides the Coral8 Studio™ which gives the capability of creating customized CEP applications to be run on the Coral8 Engine™. The Aleri platform for CEP is currently being used by several major organisations. The Platform is designed to be run on separated dedicated servers.

## 

## 3.3    Concepts of fault tolerance in distributed systems

Fault tolerance [8, 9] is the ability of a system to perform its function correctly even in the presence of internal faults. Purpose of introducing this concept is to increase the dependability of a system. So the important thing is introducing fault prevention methods which constantly inspect the system and eliminate any cause of fault creation.

### 3.3.1    Fault classification:

Faults can be grouped according to their duration, cause and behaviour [8]. Under each type there are different types of faults which are described below.

**Faults based on duration:**

* Transient faults – this type of fault ultimately vanishes without any implicit action. Intermittent fault is a problematic type of this, which recurs often.
* Persistent faults – these faults will remain, unless it is removed by some external function. The permanent faults are more severe but can be more easily handled than transient faults.

**Faults based on underlying cause:**

* Design faults – these faults are the result of design failures. Today many fault-tolerant systems are built with the assumption that design faults are inevitable. But these things can be avoided by careful design of the system.
* Operational faults – these occur during the life time of the system. It is mainly due to physical causes such as processor failures or disk crashes.

**Faults based on behaviour of failed components:**

* Crash faults – in this type of fault the component either completely stops operating or never returns to a valid state
* Omission faults – in this type of fault the component completely fails to perform its work
* Timing faults – here the component does not complete its work on time
* Byzantine faults – these are faults of an arbitrary nature. As an example a faulty processor sends different messages to different processors, to confuse them.

**Common failures occur in distributed systems:**

* Processing site failures
* Communication media failures
* Transmission delays

### 3.3.2    The following procedures have been used in distributed system design:

There are specific procedures available in distributed system design [3].

* Error detection
* Damage confinement
* Error recovery
* Fault treatment

**Error detection**

Include the mechanisms to identify whether the system is in an incorrect state. Replication checks, timing checks, run time constraint checks and diagnostic checks are a few methods available for error detecting.

**Damage confinement**

This process is to ensure that the damaged part of the system is isolated from the other components. It needs to identify flow of information along each component in the system and check errors. It determines the boundary of damaged components and isolates them from the rest, until it is correctly fixed.

**Error recovery**

Generally there are two approaches to restore the system to a valid state.

* The backward error recovery restores system, to a previously known valid state. But it is not possible since it requires constant check pointing in the system and causes lot of overheads in the total run of the system.
* Forward error recovery is a mechanism to drive system to a valid state, when an error occurs. But the kind of error needs to be known in advance, to perform this task. So this is also a difficult task to perform in practice.

**Fault treatment**

Fault treatment is the process of repairing the identified faults. Normally systems need to have integrated backup components to replace the failed components.  For the operation of backup components it needs to synchronize with the rest of the system. Generally cold standby, warm standby, hot standby are the methods used in distributed systems.

### 3.3.3 Fault tolerant algorithms

Generally there are different kinds of fault tolerant algorithms available. These algorithms address byzantine failures, stopping failures, communication failures, and resource allocation problems [8].

**Approaches for communication failures:**

The simple way to handle communication faults is to use a timeout mechanism with acknowledgments. This approach message is sent through a selected path. If the sender does not receive an acknowledgment signal from the receiver, the sender will resend the message either through the same path or through node-disjoint path.

**Approaches for stopping failures:**

In stopping failures, processes may simply stop without warning or sometimes it stops after sending its messages for some rounds but before performing its transition for that round. Flood set algorithm which is used for stopping failures has to satisfy agreements and validation tests.

## 3.4    Replication techniques

Replication techniques [10] are used in distributed systems to achieve fault tolerance. Main replication techniques used in distributed systems are described below.

### 3.4.1    Active replication

In active replication, all the replicas of the primary node, process all incoming messages concurrently, to have their internal state synchronized. If there are no failures in the system, the output can take from any of the replicate nodes.

Figure : Active Replication

Message

Process

Process

Process

Rep Node

Rep Node

Primary Node

### 3.4.2    Passive replication

In passive replication only the primary node, processes the input messages and provide the output. In the absence of faults in the system this works fine. Internal states of the other replicate nodes are kept in a way using the check pointing mechanism, such that regularly taking snapshots from the primary node to update the replicate nodes.

Figure : Passive Replication

Message

Process

Process

Process

Backup Node

Backup Node

Primary Node

### 3.4.3    Semi active replication

This is a hybrid of active and passive replication. Only one node does the processing and outputs the results. Internal states of the other replicas kept using direct processing of input messages or keep mini checkpoints.

### 3.4.4 Problems in replication techniques

* Waste of resources

In active replication all the replicate nodes process the same input as their primary node does. This can be considered wastage of processing nodes. Also in passive replication, replicas of primary node do nothing and sometimes they will standby longer times till primary node crashes.

* A better way to utilize the resources in the distributed event systems is to use all the available computing nodes for event processing. Keeping check-pointing data or keeping log records of triggers assigned can be done. And it will be more effective in distributed event processing systems. Check-pointing and log file creation will be discussed in the following topics.

## 3.5    Software Transactional Memory

## 

Software transactional memory [11] [12] is a concept originating from concurrent programming which deals with controlling concurrent access to shared memory. It is analogues to handling database transactions in any modern database system. Instead of using locks to serialize access to objects, the updates to the shared memory objects are done concurrently in the context of transactions. A transaction can be described simply as a piece of code that executes a series of reads and writes to one or more objects in the shared memory. The reads and writes of each transaction logically occur at a single instant in time (atomically). Thus intermediate states are not visible to other transactions which are executing concurrently. The concept of software transactional memory can be used on distributed systems as well to maintain consistent and up-to-date objects replicated over several nodes in the network. This approach can solve most of the data inconsistency problems which occurs due to link and process failures in modern distributed systems.

Transactions use the concept of “Versions” for objects. The version of an object is just is a counter that starts at zero and increments every time an object is modified. Each time the object is read, the version is read along with it for later comparisons. Some of the basic concepts of Software transactional memory can be described as follows.

* **Optimistic Reads**: Every time an object is read by a process, its version is also obtained. When a transaction is committed on that object, the version of that object is verified to be the same as when it was read. If the version is different, the transaction is aborted and rolled back, undoing any writes it did, and then the operation is attempted all over again. If the version has not changed the commit can go ahead. The main assumption here is that the concurrent writes aren't going to tamper with the objects we're reading very often, just other objects we don't care about.
* **Private Revocable Writes:** When an object is to be written by a process the object is locked. The lock prevents other processes from reading or writing it simultaneously, and the transaction will continue to hold this lock until it commits or aborts. Then a private copy of the object(s) is maid. Any changes to be made to the object(s) are made to this private copy. The process can atomically swap out the old copy for the updated copy when the transaction is committed. To abort the transaction, all the write locks are released and all the private copies are discarded.
* **Deadlock Detection:** Dead locks are inevitable in any scenario which uses locks. To deal with this, the program is periodically interrupted (much as a garbage collector does) and scanned for a cycle of objects that are deadlocked, each waiting on a lock held by the next. Then one of the processes is aborted, releasing its locks and breaking the cycle. With this system in place, deadlocks can never occur.

* **Edge Cases:** If a transaction which has read inconsistent state tries to commit, it will fail, but it's also possible that such a transaction could go into an infinite loop or have a fatal exception because it wasn't written to deal with such a situation. Again, the runtime is responsible for detecting and aborting such transactions.

The software transactional memory that was used in this project is called XSTM [Reference], it is an open source library which enables high performance object replication (object synchronization) between processes. It is an object oriented Distributed Shared Memory, or a Distributed Object Cache or more generally a Software Transactional Memory.

To successfully demonstrate the full functionality of XSTM it is best to consider a distributed system with two processes running on two separate machines.

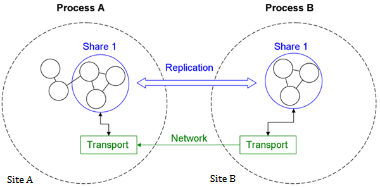


Figure : The Structure of XSTM

The two machines have two separate “Sites”, a Site is where an instance of a replicated object stands. XSTM maintains an identical graph of objects on each site. If a field of an object is modified, it is modified on the other side with a transaction. To define which objects must be replicated and which ones are local, XSTM defines Shares. A Share is a set of objects which define the boundary of the replicated graph. All the objects which are added to a share are replicated through all the shares which are subscribed to it.

## 3.6    Check pointing techniques

Check pointing mechanisms [13, 14] are there to provide fault tolerance to the distributed systems. Check pointing mechanism is one of the roll back recovery approaches used in distributed systems.  The goal of any roll back recovery approach is to bring the system into a consistent state when inconsistencies occur due to failures.

Checkpoints can be taken in different ways, one is computing nodes in the distributed system can store their register values to the stable storage or dedicated checkpoint server. Another method is to checkpoint event id or event triggers which come to a particular computing node. So if any failure happens to the computing node, when it restarts its last known successful state can be restored using the checkpoints.

When it comes to addressing problems of check-pointing mechanisms roll-back propagation is a big issue. It means rolling back a particular process cause one or more processes, to roll back. Finally this may end up with loss of a large amount of useful computing works and the system can be rolled back to the initial state of computation.

Overheads caused by the check pointing process to the overall system performance also need to be considered here. Normally when taking checkpoints most of the time it needs to stop the current working process of the computing nodes, eventually when taking checkpoints of the whole system the  performance can be degraded and it needs to be addressed effectively.

Following is a description of some of the check-pointing mechanisms available.

### 3.6.1    Coordinated check-pointing

In coordinated check-pointing there should be a process to coordinate the checkpoints of the computing nodes in order to form a consistent global state. There are different ways to take checkpoints using this mechanism. If perfectly synchronized clocks are available at processes, then all processes agree at what instant of time they will take checkpoints and clocks at processes trigger the local check-pointing actions, at all processes. But it is difficult to find perfectly synchronized clocks, therefore to guarantee the checkpoint consistency one option is to use blocking or non-blocking algorithms to collect checkpoints.

Coordinated check-pointing provide an easy way to recover from failure because it does not suffer from rollback propagations. It also minimizes storage overheads since only one checkpoint need be stored in the stable storage.

Coordinated checkpoint schemes suffer from the large latency involved in saving the checkpoints since a consistent checkpoint needs to be determined before the checkpoints can be written to stable storage.

### 3.6.2    Uncoordinated check-pointing

This is another scheme to store checkpoint data in distributed systems. Here each process independently saves its state. During restart, these processes search the set of saved checkpoints for a consistent state from which the execution can resume again. The main advantage of this scheme is that a checkpoint can take place when it is most convenient. As an example a process may perform checkpoints when the state of the process is small.

But the problem associated with uncoordinated check-pointing is, it is susceptible to rollback propagation which was described earlier. Another problem is that a process can take checkpoints that will never be part of consistent global state. Also in uncoordinated check-pointing each process keep multiple checkpoints and results in a large amount of storage overhead.

### 3.6.3    Disk-less check-pointing

Disk-less check-pointing is an alternative approach to disk-based check-pointing. In this method it takes a snapshot of the state of a program on a distributed system without relying on stable storage. It replaces stable storage with memory and processor redundancy. But the problem in disk-less check-pointing is, it requires high memory overheads for storing checkpoints.

### 3.6.4    Communication induced check-pointing

In this method it allows the processes to take some of their checkpoints independently. Here it tries to prevent the domino effect by forcing the processors to take additional checkpoints based on protocol related information, associated on the application messages, it receives from other processors. Taking additional checkpoints is important in determining a consistent global state. However, the forced checkpoint must be taken before the application may process the contents of the message, possibly leading to high latency and overheads.

## 3.7    Log based rollback recovery

Another roll-back recovery approach used in the distributed systems is keeping log records of the events occur in the distributed systems [8]. Log based rollback recovery assumes that all non deterministic events can be identified and their corresponding determinants can be logged to stable storage. After failure occurs, the failed processes recover by using the logged determinants to replay the corresponding non deterministic events correctly as they occurred during the pre-failure execution.

There are three types of log based rollback recovery protocols.

* Pessimistic logging – this guarantees that orphans are never created due to a failure. This can be used to simplify the recovery process, garbage collection and output commit. Orphans are created when a failed process sent a message during any state, which cannot recover and the receiver becomes an orphan process.
* Optimistic logging – this mechanism reduces the failure free performance overhead but it allow orphans to be created due to failures. Because of this orphans recovery, garbage collection and output commit is difficult to achieve.
* Causal logging – this attempt to combine the advantages of low performance overheads and fast output commits, but they require complex recovery and garbage collection.

## 3.8 Stratification

Stratification is a concept which can be used to modularize CEP systems, which typically simplifies their design and also provides enhanced scalability as an added advantage. There has been a researches [15-17] carried out by researches on this area.

The term stratification, which is derived from earth resource engineering, refers to the establishment of layers. In this technique [15], each event processing definition is mapped to a specific event processing agent type. Then a graph is created which contains event processing agent types (vertices of the graph) and their connections (edges of the graph). Raw incoming events to the system are represented by a edge which has only a target vertex. Similarly, derived event emitted by the system to the outside is represented by an edge without target vertex but with a source vertex. After defining the graph as above, it is subjected to certain modifications to eliminate redundancy and also to make it optimal for processing.

After above process taking place, the actual stratification algorithm is applied. It starts by first finding independent sub graphs within the event processing dependency graph. Then the stratification algorithm converts each such sub graph to an event processing sub-network by assigning and event processing agent for each event processing operation type.

Then again the algorithm examines the event processing graph to identify the sub-graphs with strict independence (which refers to the sub graphs with no connecting edges). Such sub graphs are then turned to be stratum levels. With this operation, the event processing network becomes completely stratified.

## 3.9 Scalability and load distribution in distributed systems

Scalability defines the ability of distributed system to handle growing amount of work in an efficient and effective manner. Simply it refers to the capability of a distributed system to increase its throughput after new resources added to the system. Resources include adding new computing nodes to the distributed event processing system.

Following are few of the dimensions to measure the scalability of a distributed system.

* **Load scalability** – it refers to the ability of distributed system expand easily and to accommodate the increase or decrease of loads without any performance degrade. Otherwise how easy to modified, added or remove computing node or a node partition in the system to accommodate changing load.
* **Functional scalability** – it is the ability to improve the distributed system by adding new functionality at minimal cost.

We can also identify two major scaling applied in distributed systems:

* **Scaling-up (vertical scalability)** - refers to the improvement of the processing capability of nodes.
* **Scaling-out (horizontal scalability)** - refers to adding new nodes. This technique is found to yield better performance in high-throughput applications than scale-up [18]. However, due to the distribution of processing, this causes the system to have less fault-tolerance.

Load distribution is the process of transferring units of work among processing elements during execution to maintain balance across processing elements.

Desirable properties of load distribution algorithms for CEP systems are as follows [15]:

* **Dynamic:**

Dynamic load distribution which is also called load balancing, assumes little or no compile-time knowledge about the runtime parameters such as task execution times or communication delays. Dynamic load balancing does the runtime redistribution of processes to achieve performance goals. It improves the system performance by providing a better dynamic utilization of all resources in the entire system.

* **Consideration of semantics of node dependencies:**

In circumstances where load is migrated from a node to another node, it is desirable for the algorithm to consider node dependencies such as whether it is possible for the potential new node to reach the nodes which distribute required input events and also the nodes which consume the derived events.

* **Link latencies:**

It is desirable for the nodes to be positioned in a way that reduces unwanted communication latencies.

* **Splitting of event processing agents:**

Rather than moving an entire agent to another node with more processing capability, the algorithm should be able to split it and distribute the workload among multiple nodes.

* **Monitoring load statistics:**

For algorithm to be efficient, it is desirable for it to have the ability to take load statistics into account such as mean and variance of the frequency of the incoming and derived events.

* **Consideration of stateful nature**

Since the event processing agents of a CEP system should maintain states, (which are required to identify patterns among events) the load distribution algorithm should be able to preserve an agent’s state even when it is migrated or splitted.

One of the approaches we studied to achieve scalability in distributed event processing systems is with the aid of stratification we discussed under modularizing EP and CEP systems [15]. The original idea is on scaling event processing systems, but the same idea can be extended for complex event processing systems too.

After partitioning the event processing dependency graph into strata, it can then be used for maximizing throughput of the application. Since the analysis of event dependency graph allows identify relationships between agents and a stratum can have similar type of agents, the events can be processed in parallel on all agents (running on all nodes in that stratum) in that stratum. The researchers also suggest a load distribution algorithm by analysis of synthetic and real workloads which has many of the above discussed characteristics desirable for an effective and efficient load distribution algorithm. However, it is still at a conceptual stage and can be difficult to implement for practical usage.

## 3.10 Leader in Distributed Systems

If we consider our day to day normal life, in many cases we have leaders. Let us imagine a class room of boy school. In the class room there are students who are physically very much similar to each other. They all have the same body parts and most of them have the same set of skills as well. In class room we elect leader to carry out special tasks like coordinating students to work as a group, coordinating students and teachers, etc. Any student can become the leader and perform all these tasks.

So now we look at a distributed system. In a distributed system, all most identical processes running in different hosts but they have at least one unique feature. As an example, a unique ID for the host or network IP address. So, all these processes can perform the same work such as processing data. But in order to perform efficiently and effectively, all these processes need to be coordinated. So in distributed system also we need a leader to perform these coordination tasks. Hence we elect a leader among the processes in a distributed system. Leader of a distributed system is normally performing as a coordinator, initiator, sequencer, etc [19].

## 3.11 Election Algorithms for Distributed Systems

In order to elect a leader for a distributed system there are standard set of algorithms we can use. These algorithms are used in different circumstances, backgrounds and we cannot use every algorithm in every scenario. So depending on the architecture and design of the distributed system, we select an algorithm to use. The designer of this algorithm has to prove the ability of the algorithm to work under a given circumstance and it elects only one leader at any given time. So the standard leader election algorithms are categorized based on several parameters such as network topology, synchronous/asynchronous behaviour of the distributed system and we select an algorithm from a category based on the same set of parameters.

The first classification is based on the network topology. The two categories are arbitrary network and ring networks. Token ring networks were used in the early development of the Ethernet and other Local Area Networking concepts. The algorithms introduced to use in these type of network are considered as Ring based Election Algorithms. These algorithms highly depend on the network topology (the ring) in its operation. As an example the simplest ring based election algorithm, LCR which was developed by Le Lann, Chang and Roberts. This algorithm assumes the ring is unidirectional and it does not rely on the number of nodes/hosts in the network [19]. We can use ring based algorithms in an arbitrary network as well. We create a virtual ring in the arbitrary network and then we use the virtual ring to execute the algorithm. In this manner we can use any of the ring based algorithms in an arbitrary network as well. Arbitrary network based algorithms means the algorithms which are not depending on the network topology for its operations and therefore can be used in any network. As an example we can consider the FloodMax algorithm. It can be used in any network, even in a ring network.

The next classification is based on the Synchronous and Asynchronous behaviour of the distributed system. The algorithms are categorized, modified depending on the synchronous and asynchronous behaviour of the network as well. As an example there are two slightly different versions of LCR ring based election algorithm for synchronous systems and asynchronous systems. A synchronous network system consists of a collection of computing elements located at the nodes of a directed network graph [20]. A system of parallel processes is said to be synchronous if all processes run using the same clock [21]. Examples of synchronous systems are certain large centralized multiprocessing computers and VLSI chips containing many separate parallel processing elements. In a synchronous distributed system, each and every process has a state at any given time and the state transitions happen with messages [20]. In synchronous systems after processing a certain message in the queue, it will change the state according to the transition function defined for the system. Then it deletes and discards all the other messages in the channel. This is the main difference with the Asynchronous systems. It is said to be asynchronousif each process has its own independent clock. Examples of asynchronous systems are distributed computer networks and I/O systems for conventional computers [21]. Asynchronous systems are modelled by Input/output Automaton model [20]. This model is successfully used to model both asynchronous shared memory systems and asynchronous network systems. In asynchronous system broadcasting and multicasting techniques are widely used and after processing a message other messages in the channel are not discarded.

## 3.12 Dynamic Service Discovery

Service discovery is a concept which evolved with the Service Oriented Architecture (SOA). Service discovery allows automatic detection of devices and services offered by these devices on a computer network [22]. This service discovery allows software to make changes to themselves while the software is already running. Hence the availability and reliability of the system will not be affected by the changes of the software runtime parameters such as input load. Service discovery is widely used with Web service and OSGi Remote Services.

In Web Services (WS) and Remote Services (RS), there are 3 major components in a system. They are Service Provider, Service Client and Service Registry [23]. Service Registry is a map between services and service providers. In this map there is a unique entry for each service and that entry comprises the service provider as well. One service provider may provide more than one service. Hence in the map, service name is the key and the service provider is the value. Service registry works as a lookup service. Service clients can search for the service provider using the service name as the key. Service Provider is the owner of the service and service provider has to register the service with a unique name in the Service Registry in order to provide service to clients. Service Client is the service user and if the user need to access a service provided by a service provider, first the client has to locate the service registry. After locating the correct Service Registry, then the client has to query the service registry with the service name to find the service provider. These are the general and common procedures of accessing a SOAP based Web Service, a OSGi Remote Service or a Java Remote Method Invocation (RMI) Service.

There are 2 steps in service discovery process. First step is locating the service registry such as a name service or a directory service. E.g.: DNS, Registry Server, RMI Registry. There are 2 possible methods we can use to locate the Registry. First method is to use static preconfigured information about the registry. E.g.: DNS and Registry Server. In this case service user has prior knowledge about the lookup service such as DNS server address. Other method is used if the service client does not the information about the location of the lookup service. Then first client has to search for the lookup service, locate the lookup service and then do the lookup for the service provider using the service name. In such situations we use dynamic service discovery mechanism using IP multicasting, broadcasting so that client can find the lookup service by listening to the multicast and broadcast messages of their network. This method is considered as a form of Dynamic Service Discovery as most of the service details are acquired during the runtime. E.g.: Registry Server IP address. Second and the last step in service discovery process is, query/lookup the Service Registry (Name Service, Directory Service) for the desired service name and service provider. In this case also it is possible to do it in 2 methods. First method is the client has a prior knowledge of the required service name, so that the client directly lookup the service registry to find the service provider. If the client does not have the knowledge on the service name, then client cannot find the service. In this case also, we use IP Multicasting and Broadcasting to provide information about the existing services in the service registry. Then the client can listen to the multicast and broadcast messages and find the service name then make the lookup in the service registry. This also considered as a form of Dynamic Service Discovery. This method is widely used with Java RMI Services as Java RMI Service Providers usually use the same host to provide both Java RMI Service as RMI Server and Java RMI Registry Service.

If we use the same host as the Service Provider and Service Registry, when we use the second type of Dynamic Service Discovery where we let the others know about our Services, that information carries the Service Registry location embedded in that message. Therefore knowledge of the Services provided by a provider implies knowledge of the both Service Registry Location and its registered Services. This method is very much effective with Java RMI Services.

Dynamic Service Discovery can be used to provide scalability to a distributed system. Let us assume that the each node has a service registry running and the services running in a node are registered in the service registry running in the same node. When the load on a distributed system is higher than it can handle at a given time, then the system needs more components of the same component type to handle the extra load. So we start new node of the same component type and then broadcast and/or multicast the services provided by the new node. Then the service proxy and other nodes can identify the new node and start forwarding the load to the new node. In this case the service providers use the discovered service names via multicasting and/or broadcasting to access the services running in a node. Using this method we can scale up the system anytime we need.

# **Chapter 4 - Project Design Overview**

## 4.1 Introduction

Under this topic we discuss about the system architecture and its major components. The objective of this chapter is to enlighten the reader on the exact functionality of all the major components of epZilla.

In obtaining the major project objectives, we went through a lot of research papers and other materials. Hence we were able to clearly identify most of the problems faced while designing a scalable and reliable distributed system. During our research we were also able to identify possible solutions to those problems as well. In our effort to create a scalable fault tolerant architecture we were able to combine the knowledge we gathered, in coming up with several draft architectures which were possible final solutions to the problem at hand.

## 4.2 System Architecture

High level system architecture of the project epZilla is shown in the following figure. Further implementation details of the system components and the integration parts of it are discussed under **Chapter 5**.

Following terms used in the following figure.

* **STM**- (Distributed) Software Transactional Memory
* Parent node in the node cluster is shown in red colour.

Event

Dispatcher

Event

Dispatcher

Events

N11

N13

N12

N14

Accumulator 2

Node cluster 1

Node cluster 2

Alert

Client

Feedback to event dispatchers

N21

N22

N23

N24

Allow to add new node clusters for scalability

STM

STM

STM

STM

Name Server

Event

Dispatcher

Accumulator 1

...

...

...

Figure : System Architecture

## 4.3 Client side operation

Following flow diagram shows the client initialization and the event flow process. Initial process is to Look up the Name Server and get details of the Dispatcher services avaialable. If the process succeeds, then the client can initialize the Dispatcher service and start sending triggers and events. After the processing the event against the available triggers, the notification is sent back to the client.

Figure : Event flow of the System

Initialize Dispatcher

Name Server Lookup

Valid

Send Event to Dispatcher

Event processing

[Success]

End

Start

Notification

Details

# 

# Chapter 5 - System Implementation

The objective of this chapter is to cover all the implementation details of the project epZilla comprehensively. Also this chapter provides highly detailed technical perspective of the algorithms and concepts used and the rationalizations behind them. In the first sections (5.1 to 5.5) we shall look at the various components that make up the epZilla architecture and then in the rest of the sections (5.6 to 5.12) actually go the details about the technologies that were used in their development.

## 5.1 epZilla Client

Project epZilla is implemented to support multiple clients. A client can be an organization or a component of an organization or a single event source. The client application facilitates the sending of both events and triggers and also accepts the results of the actual event processing. Since there can be multiple of such clients, to identify each client uniquely we put a *client id* in each and every *trigger* and *event* send by a particular client. The *client id* is derived from the IP address of the client, because it is a promising solution in every aspect. We add the generated client id as a separate tag to the each event sent by the client. This *client id* is very useful when sending the call-backs to the clients. In our system we refer to this phenomenon as client call-back.

We developed a custom Event and Trigger generator, for our testing environment. By using that generator client can send events to the Dispatcher unit at a speed 500 events/sec. We use RMI for all the communication channels and use proper interfaces in every component.

GUI parts of the epZilla Client is implemented using java swing controllers. In the development process we use singleton design pattern and the concepts of MVC design pattern.

## 5.2 Name Server

The Name Server is developed using RMI, data structures and collections available in java.util package. Also we always use appropriate OOP concepts in the development process. Since Name Server does not perform lot of critical processing we use single computer without any replications. We also added XML file to keep details of the registered Dispatchers. So if any failure occurred we can use the XML file to recover the details of the Dispatchers.

We use generic data structures like Vectors, Hash tables to keep dispatcher details, since we need to make sure system is highly scalable and fault tolerant. Because of this any number of Dispatchers can be added and multiple clients can connect to the Name Server same time.

Another important function added to the Name Server is the load balancing of the Dispatchers. This approach uses a combination of both round robin and weighted load balancing mechanisms. This works as follows.

When a Dispatcher initially registered in the Name Server, load factor of it is assigned to zero. When client perform the lookup operation in the Name Server, it always return the details of the least loaded Dispatcher to the client. Following code snapshot shows how to compare the load of the Dispatchers. We use Hash Map to store the Dispatcher IP and its load. As mentioned earlier when a client connects to a Dispatcher its load factor increment and the details are stored/updated in the Map. So when client perform the Look up operation it compare the all the Dispatchers and give the least loaded Dispatcher IP to the client.

public int compare(Object o1, Object o2) {

if (((Integer) ((Map.Entry) o1).getValue()).intValue() > ((Integer) ((Map.Entry) o2).getValue()).intValue()) {

return (1);

} else if (((Integer) ((Map.Entry) o1).getValue()).intValue() < ((Integer) ((Map.Entry) o2).getValue()).intValue()) {

return (-1);

} else {

return (0);

}

}

Figure : IP Comparator algorithm

After when client connect with the Dispatcher, it notifies the Name Server that there is a client connected to it. The Name Server then increases particular Dispatchers load factor by one unit. Also when a client disconnects from the Dispatcher, it again notifies the Name Server to decrease the load. Data Structures like Arrays, Hash tables and Comparators are used to design the Load balancer class. Main reasons to select the above mentioned data structures are to support concurrency of the system.

There is composition between the Name Server implementation class and the Load Balancer class to ensure it works in a proper way. All the functionalities of the Load Balancer are depending upon the Name Server.

## 5.3 Dispatcher

Dispatcher is one of the most critical components of the system. RMI and STM are the major technologies use to develop dispatcher unit. Following figure shows the high level architecture of a single Dispatcher unit.

User Interface

Distribution Logic

Data manager

STM

RMI

Figure : Dispatcher overview

Each Dispatcher has properly defined interfaces which communicate using RMI. We arranged the source files of Dispatcher into several packages to make sure that the dependencies make sense. We also use OOP concepts like association, composition, inheritance, interfaces, in every possible place. Also we make use of some major design patterns in the development process. MVC pattern is used in the development of the user interface of Dispatcher and singleton pattern in appropriate places.

Following is a comprehensive description of implemented functionalities in the Dispatcher component.

* The epZilla architecture is designed to have several dispatcher nodes. The actual number of dispatchers to be used is determined by the performance and reliability requirements of the user of the system.
* All the Dispatchers in the system are active. Primary and other dispatchers are connected through a Software Transactional Memory (STM). Any dispatcher accepts triggers from the client and puts them to a shared List in the STM.
* The primary dispatcher assigns the triggers to each Node cluster. The set of triggers, assigned to each cluster, is determined by the total triggers given to the system. Events received from the clients are not added to the STM. Instead all the events are routed to the registered Node Cluster leaders.
* Notification system is there to accept feedback from the accumulators. According to the feedback dispatchers send a call back to the client who sent the event .The Dispatcher keeps track of each and every trigger received using check points, which will be discussed in the sub section **5.11**.

## 5.4 Cluster Node

The cluster nodes are the most important components of the architecture. These nodes do the actual event processing. All the other nodes are just there to support the event processing functionality. The actual number of event processing cluster nodes determines the performance of the system. The dispatchers are used to efficiently route events and triggers to the cluster nodes. The cluster node contains an event processing engine which does the actual processing. This engine can be any modern event processing engine. Our implementation of the cluster node facilitates the event processing engine to successfully process a stream of events.

Each of the node clusters are encapsulated by a separate STM, this STM contains all the triggers that the dispatcher assigned for that cluster. When a new node joins the cluster it can get the trigger information just by connecting to the STM. Each node cluster has a designated leader node which is elected via a leader election algorithm. The dispatcher sends the stream of events to the leader. The leader then distributes the events among the nodes of the cluster in a round robbing way. Each cluster node processes the events that it gets against the assigned list of triggers for the cluster and sends the results to the result accumulator nodes.

During the processing of the events each node does a self calculation of its load and sends that information periodically to the leader of the node cluster. The leader does the dynamic load balancing of the cluster by adding or removing nodes to the cluster based on the load on all of its nodes.

Following figure shows how different components are arranged inside a single Node of an event processing node cluster.

Figure : Node component

User interface

Event processing logic

Data manager

RMI

STM

Event processor

## 5.5 Accumulator

The epZilla architecture can have any number of result accumulators depending on the event processing load and the desired level of fault tolerance. Accumulator is responsible for accepting partial events, what we refer to as derived events in this context. The accumulator accepts the results of the actual event processing sent by the cluster nodes. Since each event is processed by several node clusters which have different triggers assigned to them, the accumulators wait till all the clusters have sent the processed results and then build a total result per single event and send that result back to the client. Each event is sent to at least two accumulators to achieve fault tolerance in case of a single dispatcher failure. There is a Notification system implemented in the Accumulator which can send the generated Alerts to the relevant Client. We use proper interface, to provide the functionalities to the outside. Also use fair amount of design patterns like observers, singleton, MVC and other best practices in the development process. Following figure shows how the components inside the Accumulator are organized.

Figure : Accumulator Overview

Event accumulator

Accumulation Logic

Data manager

STM

RMI

Notification System

## 5.6 Software Transactional Memory

The usage of the Software Transactional Memory (STM) is one of the major design features of the project. This project is the first documented project to use a STM for the purpose of achieving fault tolerance. The STM is used on three occasions inside the architecture.

### 5.6.2 Dispatcher STM Implementation

The dispatcher STM implementation manages the scalability and fault tolerance of the dispatchers of the system. The STM synchronizes the critical operational data among the set of dispatchers. The data synchronized by the STM include

* The list of triggers and their cluster assignment details.
* The list of client details.
* The list of registered cluster leader details.

When the systems initializes, one of the dynamically discovered dispatchers is elected as the leader through the leader election algorithm. The elected leader starts as a STM server and initializes the STM. The other dispatchers connect to the STM server as clients. The STM server adds the lists of data to be synchronized among the available dispatchers. The following code section found in the “DispatcherAsServer” class adds the list of triggers to the STM.

Figure : Add list of triggers to STM

public static void loadTriggers() {

DispatcherUIController.appendTextToStatus("Shared Transacted list Added for Triggers..");

if (Site.getLocal().getPendingCommitCount() < Site.MAX\_PENDING\_COMMIT\_COUNT) {

Site.getLocal().allowThread();

Transaction transaction = Site.getLocal().startTransaction();

share.add(TriggerManager.getTriggers()); // add list of triggers to the share

transaction.commit();

}

}

The list of triggers is empty when it is first added. Then when the system is running, triggers can be added by any of the dispatchers as they are received from the clients. The following method adds a trigger to the list which is shared on the STM via committing a transaction.

public static boolean addTriggerToList(String trigger, String clientID) {

boolean success = false;

if (getTriggers() != null) {

synchronized (triggerIdSyncLock) {

if (Site.getLocal().getPendingCommitCount() < Site.MAX\_PENDING\_COMMIT\_COUNT) {

Site.getLocal().allowThread();

Transaction transaction = Site.getLocal().startTransaction(); //start the transaction

TriggerInfoObject obj = new TriggerInfoObject();

// ID is the sequential number of the trigger

obj.settriggerID("TID:" + String.valueOf(count));

obj.setclientID(clientID);

obj.settrigger(new String(trigger));

getTriggers().add(obj); // add trigger to the shared list

sendTriggersToclusters(trigger);

transaction.commit(); //commit the transaction

success = true;

System.out.println(trigger);

}

if (success) {

count++;

}

}

}

return success;

}

Figure : Add triggers to shared list

The usage of transactions assures the complete synchronization of all the dispatchers.

### 5.6.3 Cluster and Accumulator STM Implementation

The STM implementation in the node cluster and the accumulator are based on the same concepts as in the dispatcher. The cluster STM is used to synchronize the following data among the nodes of a given cluster

* The list of triggers assigned to that cluster.
* The list of node IPs of that cluster.
* The average performance data of all the nodes in the cluster.

The STM of the result accumulator is used to synchronize the following data among all the available accumulator nodes

* Accumulator node IP list.
* List of event Ids.
* List of cluster leader details.

We were extremely concerned on what data to store on the STM. The STM is a highly network dependent resource and we found out that we need to control its usage in a calculated way to have the maximum performance of the system together with the highest possible availability. Only the data crucial to the continuous operation of the system was stored in the STM.

## 5.7 Stratification and trigger distribution

The epZilla architecture can have several clusters where nodes are connected and synchronized through a software transactional memory. Upon receipt of triggers to dispatchers, each dispatcher dynamically assigns a cluster to each trigger by looking at its dependencies with the existing triggers. The algorithm is implemented targeting better performance than running the entire dependency analysis process and approximately assigns a cluster for each query.

The strategy of assigning a cluster to a trigger is done conditionally based on two conditions:

1. The trigger has dependencies with other triggers in a existing cluster – in this scenario, the cluster which the trigger should be assigned to is obvious and the trigger is immediately assigned to the cluster and the dependency maps are updated with the metadata of the assigned trigger.
2. The trigger doesn’t have dependencies with other triggers in clusters – in this scenario, we use two different methods for assigning triggers to clusters based on conditions.
3. The system doesn’t have a large enough load and enough number of triggers assigned to clusters – this situation occurs when the system has just started operation, or when there’s not enough triggers assigned or when a large number of triggers have been withdrawn from the system. In this case, in order to achieve an even distribution of triggers, we make use of round robin approach without considering load.
4. The system has enough load and enough triggers assigned for clusters - in this scenario, instead of using round robin approach for assigning a cluster to a trigger, it is done by examining loads at different clusters. This approach allows each cluster to have an approximately even load in the long run.

### 5.7.1 Implementation details of dynamic dependency analysis

We make use of a software transactional memory approach for sharing the dependencies of triggers among the set of available dispatchers, which makes it possible that each trigger is analyzed for its dependencies globally and to be correctly assigned to a cluster.

The dependencies are stored as sets, each set mapping to a cluster and a stratum consists of collection of such sets. Upon receipt of a trigger, its input requirements are compared against the dependency collections (each mapping to a stratum or a part of a stratum), which allows indentifying of the correct stratum for the trigger. Once the trigger’s stratum is identified, then the process to identify its cluster starts running. It’s done by comparing the state retaining requirements with existing clusters. By the end of this process, the correct cluster for the trigger is identified given that the trigger has dependencies with existing triggers.

When a trigger which doesn’t depend on existing triggers arrives, the process deviates from the above. In such situations, it is necessary to assign the trigger to a cluster considering the loads of clusters in a way that a subset of clusters won’t get overloaded. It’s possible that there can be triggers with dependencies with such a trigger may arrive lately, so assigning such queries evenly in clusters can prevent clusters from overloading.

Once the system is just started, it’s most likely that the load of the system can’t be measured properly. In this case, the triggers with no dependencies are dispatched in a round robin fashion in the system to ensure even distribution as outlined in the previous subsection. Once the system is loaded adequately, it is more desirable to assign such queries by taking into account the loads in different clusters to ensure even load distribution.

To track the dependencies, we make use of a composite structure of lists and sets which contain input and output dependencies with state retaining information. To guarantee the presence and synchronization of such dependency structures, we share all such dependency structures with all the dispatchers using the Software Transactional memory.

However, under certain situations, above approaches can cause some redundancies as well, requiring some of the sliding (accumulating) windows used for complex event processing to be replicated over several clusters. To address this issue, our system periodically re-organizes its triggers, thus making the trigger distribution more event and efficient.

Figure : Dependency Graph

Strata List

Statum 1 -Cluster List

Statum 2 -Cluster List

Statum N -Cluster List

### 5.7.2 Trigger base re-organization

The technique described above is for dynamically assigning a stratum and a cluster immediately when a trigger is arrived. However, in the long run, when triggers are added and removed randomly, it can cause different clusters to have vastly different loads, which creates the need for the system to have a mechanism for self re-organizing when needed.

Our approach of restructuring or re-organization is based on the approximation that processing overhead caused by various triggers on a particular event has a normal distribution which generally holds true for an event processing/complex event processing system with large number of triggers. In fact, in a large system, monitoring processing overheads for individual triggers causes a lot of processing overhead which makes it difficult to be used efficiently. While theoretically such approach would yield the system to be balanced evenly, out approach approximately distributes the load with less resource usage for monitoring activities.

Our entire process is based on the research carried out on high throughput stratified event processing systems, where dependencies of triggers are analyzed before each trigger is assigned to a stratum and a cluster. The entire process of analyzing dependencies and assigning strata/clusters to queries in this re-organization process needs to be carried out by a single node since the graph of dependencies of all triggers must be present and efficiently accessible in the place where the re-organization takes place. In our implementation the responsibility of re-organization is assigned to the leader of all dispatchers and the analysis of dependencies is carried out in the background while the system is running and then the triggers are moved from/to clusters by temporarily freezing the operation of the system.

### 5.7.3 Implementation of trigger re-organization

For trigger re-organization to happen, all trigger dependencies need to be analyzed. In our system, this happens by building a dependency graph of all triggers. Once the graph is built, it is then divided into sets which are sequentially ordered, by using a variant of Topological graph sorting algorithm. Such sequentially ordered sets are then mapped into strata, each stratum containing one or more sets.

Once the above process completes, strata are formed. The next step is to assign a cluster to each trigger. The approach we use here is to form disjoint sets without considering the number of actually available partitions or clusters in the system. Then these disjoint sets are assigned to clusters in a way they are approximately uniformly distributed.

The algorithm for defining strata is based on Topological sorting algorithm taken from [x] Introduction to Algorithms

Let L = list of sets.

S = set of all vertices

While S is non-empty

LS = new empty set

Previous = 0

Do

Previous = size(LS)

If node n of S doesn’t have incoming edges

Add n to LS

End If

While size(LS) != previous

Add LS to L

// all nodes in stratum are detected now

For each node n is LS

Remove n from S

End For

End While

Return L

In the end, the algorithm yields a list of sets, each set mapping to a virtual stratum.

## 

## 5.8 Dynamic Service Discovery

Service discovery allows automatic detection of devices and services offered by the devices on a computer network [22]. In service discovery there are two steps and we provide dynamic discovery for any step or each step. First step is to locate the Service Registry and the next step is to lookup Services registered in a Service Registry. Dynamic Service Discovery can be applied at both steps.

In the project we use Java RMI for all the communication between the components. Java RMI service comprises three components which are identical to the components in the Web Services and Remote Services. They are,

1. Java RMI Server – Service Provider
2. Java RMI Client – Service Client
3. Java RMI Registry – Service Registry

Unlike in Web Services, in Java RMI we run a Java RMI Registry in the RMI Server running node. Therefore the Java RMI Service providers register the Java RMI Services in the Service Registry running in the same host. Therefore, if we find the Java RMI Server, the Service Registry corresponding to that RMI Server is the same host. So we discover both Services and Service Registry in the same time. Dynamic Service Discovery component is designed and implemented to fulfil these requirements. It is used to discover Java RMI Services running in the network, to subscribe with the discovered RMI services and to unsubscribe from the subscribed RMI services

In our system we have different components such as Dispatchers, Nodes and Accumulators and we elect leaders for each of these components to initiate the Shared Transactional Memory (STM) which is used to enable the Fault Tolerance via Object Replication. We use JSTM framework which is a third party open source framework and it requires a leader to initiate the STM. So we need a leader for that. In our system we need a leader to coordinate the tasks distribute triggers and event, etc.

When electing a leader for dispatchers, we need to know what the nodes are in the system currently working as Dispatchers. So we need to discover the abilities of each and every node in this manner. We can use a static configuration file to track the functionalities of each node but it is not very much useful if some of the nodes in the system are currently not available. So we should be able to find who the dead nodes are and who the live nodes are in the system. Therefore we need to discover these details dynamically. So during the research we searched for the possible methods we can use to achieve this. So when we are doing the research, we found an existing method which we can use to discover RMI Services. It is called as Jini [25]. We found the mechanism used in the Jini described in an article in the Java World web site [24]. This is the graphical representation of Jini discovery mechanism.

## C:\Documents and Settings\Administrator\Desktop\jw-1121-rmi2.jpg

Figure : Jini Discovery Mechanism

In this design we found few problems to use in our project. As an example in our system we need to subscription (publisher-subscriber) between components to manage them easily using the Software Transactional Memory (STM). As an example cluster leaders need to subscribe with Dispatchers and non leader nodes in a cluster need to subscribe with the leader of that cluster. If we use the above mechanism, then our process becomes more complicated. For instance let’s consider a situation where dispatcher is the server and the cluster leader is the client. Then according to the above architecture, first leader has to send a multicast message and then the dispatcher reads and parses it and sends a TCP message with the server stub. Then to the cluster leader can use the stub and invoke the server methods. But in our case we don’t need to invoke the server methods right away. Another problem in this mechanism is, this design does not provide a way to know about the same type of components. As an example, there is no way that we can use this method to get to know about the other dispatchers for a given dispatcher. But it is very important in our system to know about the same type of components as we use this dynamic discovery to know about the live nodes out of all the possible nodes and that live node list is used in leader election. The main reason behind introducing the dynamic discovery to our system is to identify the same type of live nodes to use in the Leader Election Process. Since it cannot be achieved by this design, we designed our own version of Dynamic RMI service discovery mechanism. When designing the architecture of the dynamic discovery component we mainly focused 2 things.

1. Each and every component should be able to publish the abilities of them.
2. Each and every component should be able to identify the same type of components in the network and should be able to subscribe with a different type of component.

Here is the graphical representation of our dynamic service mechanism.

Figure : epZilla Dynamic Service Discovery Mechanism

RMI Client

RMI Server

UDP Multicast Message

TCP Unicast Message – Subscribe Message

In this mechanism we solved the problems in the Jini mechanism by making few changes to the design. Let’s consider RMI server is the dispatcher and client is our cluster leader. In this case cluster leader can subscribe to the dispatcher by sending a TCP message. So, one requirement can be fulfilled using this mechanism. Since dispatcher sends multicast messages stating its capabilities, other dispatchers can receive the multicast message and identify this node as a dispatcher. Each and every dispatcher can identify all the other dispatchers in this mechanism. Cluster leader sends another multicast message and nodes in that cluster can identify the leader node and register with it. All the nodes can identify all the other nodes. So this mechanism is very much successful approach to our problem of dynamic discovery. Here is the abstract view of the Dynamic Service Discovery API we have designed.

*IService*

*Publisher*

*IService*

*Locator*

Multicast Listener

Unicast Listener

Multicast Sender

Unicast Sender

Figure : Abstract View of the Dynamic Discovery API

IServicePublisher and IServiceLocator are in the lowest abstraction layer in the Service Discovery API and they are the basic interfaces of the epZilla Dynamic Service Discovery API. IServicePublisher provides abstraction to publishing the service in a very general way. Here is the declaration of IServicePublisher.

Figure : Code Segment Interface IService Publisher

/\*\*

\* This interface is used to publish services via ip multicasting.

\* Provide ability to add and remove subscribers to services for the service providers.

\* Applications have to implement this interface according to the requirements.

\* **@author** Administrator

\*

\*/

**public** **interface** IServicePublisher {

**public** **boolean** publishService();

**public** **boolean** addSubscription(String serviceClient, String serviceName);

**public** **boolean** removeSubscrition(String serviceClient, String serviceName);

}

Here is the complete implementation of the IServicePublisher interface in the DispatcherPublisher class.

Figure : Code Segment for DispatcherPubliser Implementation

package org.epzilla.common.discovery.dispatcher;

import java.util.HashSet;

import java.util.Hashtable;

import org.epzilla.common.discovery.Constants;

import org.epzilla.common.discovery.IServicePublisher;

import org.epzilla.common.discovery.multicast.\*;

public class DispatcherPublisher implements IServicePublisher {

private String serviceName="DISPATCHER\_SERVICE";

private String multicastGroupIp="224.0.0.2";

private int multicastPort=5005;

private Hashtable<Integer, String> clusterLeaderIp=new Hashtable<Integer, String>();

private HashSet<String> dispatcherList=new HashSet<String>();

public DispatcherPublisher() {

}

public boolean addSubscription(String serviceClient, String serviceName) {

if(serviceName.equalsIgnoreCase("SUBSCRIBE\_"+this.serviceName)){

synchronized (clusterLeaderIp) {

String []arr=serviceClient.split(Constants.DISPATCHER\_CLIENT\_DELIMITER);

clusterLeaderIp.put(Integer.parseInt(arr[0]), arr[1]);

return true;

}

}

return false;

}

public boolean publishService() {

MulticastSender broadcaster=new MulticastSender(multicastGroupIp,multicastPort);

broadcaster.broadcastMessage(serviceName);

return true;

}

**public** **boolean** removeSubscrition(String serviceClient, String serviceName) {

**if**(serviceName.equalsIgnoreCase("UNSUBSCRIBE\_"+**this**.serviceName)){

**synchronized** (clusterLeaderIp){

clusterLeaderIp.remove(Integer.*parseInt*(serviceClient.split(Constants.*DISPATCHER\_CLIENT\_DELIMITER*)[0]));

**return** **true**;

}

}

**return** **false**;

}

**public** **boolean** insertDispatcher(String dispatcherIp){

**synchronized** (dispatcherList) {

dispatcherList.add(dispatcherIp);

**return** **true**;

}

}

**public** **boolean** removeDispatcher(String dispatcherIp){

**synchronized** (dispatcherList) {

dispatcherList.remove(dispatcherIp);

**return** **true**;

}

}

**public** Hashtable<Integer, String> getSubscribers(){

**return** clusterLeaderIp;

}

**public** HashSet<String> getDispatchers(){

**return** dispatcherList;

}

}

Figure : Code Segment DispatcherPubliser (Cont.)

public boolean publishService() is used to publish the Services by the implementations. As an example here is the implementation of that method in the DispatcherPublisher.

**public** **boolean** publishService() {

MulticastSender broadcaster=**new** multicastSender(multicastGroupIp,multicastPort);

broadcaster.broadcastMessage(serviceName);

**return** **true**;

}

Figure : Code segment for publisherService implmentation.

public boolean addSubscription(String serviceClient, String serviceName)is used to subscribe to a given RMI service. As an example see how this is implemented in the DispatcherPublisher.

Figure : Code segment for addSubscription implementation

**public** **boolean** addSubscription(String serviceClient, String serviceName) {

**if**(serviceName.equalsIgnoreCase("SUBSCRIBE\_"+**this**.serviceName)){

**synchronized** (clusterLeaderIp) {

String []arr = serviceClient.split

(Constants.*DISPATCHER\_CLIENT\_DELIMITER*);

clusterLeaderIp.put(Integer.*parseInt*(arr[0]), arr[1]);

**return** **true**;

}

}

**return** **false**;

}

Here is the high level overview of the Dispatcher Service Discovery mechanism.

Figure : Dispatcher Dynamic Discovery mechanism overview

*IService*

*Publisher*

*IService*

*Locator*

Multicast Listener

Unicast Listener

Multicast Sender

Unicast Sender

Multicast Decoder

Dispatcher

Publisher

Dispatcher

Locator

Unicast Decoder

Here is the code block of the Multicast Message Decoder class of the Dispatcher Dynamic Discovery.

**package** org.epzilla.common.discovery.dispatcher;

**import** org.epzilla.common.discovery.Constants;

**public** **class** MulticastMessageDecoder **implements** Runnable {

//Handle the incoming broadcast messages from other dispatchers to elect the leader.

**private** String message;

**public** MulticastMessageDecoder(String message) {

**this**.message=message;

}

@Override

**public** **void** run() {

//0=message;1=sender

String []mcArr=message.split(Constants.*MULTICAST\_DELIMITER*);

**if**(mcArr[0].equalsIgnoreCase("DISPATCHER\_SERVICE") && !DispatcherDiscoveryManager.*getDispatcherPublisher*().getDispatchers().contains(mcArr[1])){

DispatcherDiscoveryManager.*getDispatcherPublisher*().insertDispatcher(mcArr[1]);

}

}

}

Figure : Code segment for MulticastMessageDecoder

As you can see this class will update the list of Dispatchers when it receives a multicast message from another dispatcher.

Here is the TCP message decoder class implementation for the Dispatcher dynamic Discovery.

**package** org.epzilla.common.discovery.dispatcher;

**import** org.epzilla.common.discovery.Constants;

**public** **class** TCPMessageDecoder **implements** Runnable {

**private** String message;

**public** TCPMessageDecoder(String message) {

**this**.message=message;

}

Figure : Code segment for TCPMessageDecoder

@Override

**public** **void** run() {

//0=message,1=ip

String []tcpArr=message.split(Constants.*TCP\_UNICAST\_DELIMITER*);

//0=cluster id,1=service name

String []arr=tcpArr[0].split(Constants.*DISPATCHER\_CLIENT\_DELIMITER*);

**if**(arr[1].equalsIgnoreCase("SUBSCRIBE\_DISPATCHER\_SERVICE")){

DispatcherDiscoveryManager.*getDispatcherPublisher*().addSubscription(arr[0]+Constants.*DISPATCHER\_CLIENT\_DELIMITER*+tcpArr[1], arr[1]);

}**else** **if**(arr[1].equalsIgnoreCase("UNSUBSCRIBE\_DISPATCHER\_SERVICE")){

DispatcherDiscoveryManager.*getDispatcherPublisher*().removeSubscrition(arr[0]+Constants.*DISPATCHER\_CLIENT\_DELIMITER*+tcpArr[1], arr[1]);

}

}

}

Figure : Code segment for TCPMessageDecoder (Cont.)

As you can see in this code block, TCP decoder updates the list of cluster leaders when it receives a TCP message asking to subscribe to the dispatcher service (“SUBSCRIBE\_DISPATCHER\_SERVICE”). It also removes anode from the cluster leader list when it receives the unsubscribe message, (“UNSUBSCRIBE\_DISPATCHER\_SERVICE”). Multicast listener, TCP listener and Multicast sender services are started when the Dispatcher Discovery Service is initiated. Node Dynamic Discovery also implemented in the code base to support the above functionalities. Dynamic Discovery services are accessed in the Leader Election component of the system as we have explained in the above section.

## 5.9 Leader Election

In this project we have several component types such as Dispatcher, Nodes and Accumulator. So it is possible that a failure in one of these components will break the complete system down. So we need to avoid such single point of failures occur in the system. For that we need to have fault tolerance in the system. The method of achieving FT is having multiple copies of same component and all these copies should be synchronized. There are 2 methods to achieve synchronization in distributed systems.

1. Message Passing
2. Shared Memory

In our project we use Shared memory concept and the method we are using is called the “Shared Transaction Memory”. The concept behind this method is called Object Replication. We use a third party framework which is called as JSTM as our STM handler. STM requires a leader to initiate the STM between same types of components. So we decided to implement a leader election mechanism for our system.

When considering the Leader Election algorithms, we have to decide which algorithm to use according to our necessities. In this case we looked at several options such as LCR (Le Lann, Chang and Roberts), FloodMax algorithms, etc. Finally we decided to use LCR algorithms as it is very simple algorithm to implement and it can produce the leader quickly as it has a time complexity of n rounds and message complexity of O(n2). Though it uses a higher number of messages, time complexity is low due to the simplicity of the algorithm.

Before explaining the LCR algorithm or its implementation, there are few important properties to understand. It is very important to understand that absolutely identical processes cannot elect a leader [20]. So we need at least one unique property in each process. In this LCR algorithm we use UID which stands for the unique id for each process. Each process is in a distinct state at any given time and there exist a finite set of states possible. Transitions between states happen according to the received and processed message to any given process.

LCR algorithm is for ring networks and it does not depend on the size of the network. It assumes the network communication unidirectional. But today, we don’t have specific networks such as Ring networks and the communication is bidirectional. So in this case we create a clockwise virtual ring and use that virtual ring to execute LCR algorithm. Formal definition of LCR algorithm stands as below.

The message alphabet M is exactly the set of UIDs.

For each i, the states in *statesi* consist of the following components:

u, a UID, initially i's UID

*send,* a UID or *null,* initially i's UID

*status,* with values in *{unknown, leader},* initially *unknown*

For each i, the transition function *transi* is defined by the following pseudocode:

*send* := *null*

if the incoming message is v, a UID, then

case

*v > u: send* : - v

*v = u: status* := *leader*

v < u: do nothing

endcase

Figure : Formal Definition of LCR Algorithm[20]

The implementation of LCR is started by defining the variables for UID and Status. These variables should be accessible throughout the whole Leader election process. Hence we have defined them with the static modifier.

**private** **static** **long** *UID*=0;

**private** **static** String *status*=Status.*UNKNOWN*.name();

Figure : Code segment of LCR Implementation

Possible set of States are defined as an enum as below.

**public** **enum** Status {

*UNKNOWN*, *LEADER*, *NON\_LEADER*;

}

Figure : Code segment of Status enumerator class in LCR Implementation

The next thing we have to do is defining the virtual ring. For that we use static configuration file of possible nodes. Let’s consider the leader election process of a cluster. Then we need to create a virtual ring for that cluster. We use a xml file called “EpzillaIpConfig.xml” which contains all the possible IP addresses of cluster nodes and their UIDs.

<ip uid="1">192.168.1.25</ip>

<ip uid="2">192.168.1.50</ip>

Figure : Code Segment of EpzillaIpConfig configuration file

When process starts up we read this file and set the UID variable by matching the IP addresses given. After that it set the status to UNKNOWN as it is set by default when system starts up. Then the process loads all the UID, IP pairs to a Hashtable. So if we use this Hashtable to iterate from top to bottom and when bottom reached then back to top, it works as a circular iteration. We use that concept to create the virtual ring. So each process can easily identify the next node by referring the Hashtable. But the problem is how we know whether all there nodes are live or not. In that place Dynamic Service Discovery comes to the theatre. We use initiate the Dynamic Service Discovery for the component based on the value stored in the setting file.

<cluster id="1" component="Node">

Figure : Cluster ID definition in the configurarion file.

As we considered the example of node cluster, above configuration data will be included in the configuration file and the cluster id value will change from cluster to cluster.

Now we initiate the dynamic discovery for node to detect all the other nodes in the same cluster, all the present dispatchers and cluster leader if available. So dynamic discovery gives us a list of nodes that are present and that list is used to extract the virtual ring for the network. So now the virtual ring is created. Now it is time to discuss the messaging of the Leader Election component.

We have defined a set of message Ids for all the possible messages as given below.

**public** **static** **char** *SEPARATOR* = '\u0016';

// This is to send the UID for leader election

**public** **static** **byte** *UID* = 1;

// This is the message id of the leader announcement

**public** **static** **byte** *LEADER* = 2;

// This is the message id of the non leaders

**public** **static** **byte** *NON\_LEADER* = 3;

// This is the pulse of the Leader to send to others

**public** **static** **byte** *PULSE* = 4;

// This is the Leader ping message

**public** **static** **byte** *PING\_LEADER* = 5;

// If the request cannot be completed here, send this.

**public** **static** **byte** *REQUEST\_NOT\_ACCEPTED* = 6;

Figure : Code Segment of Message Id definition.

All these messages are introduced with a perfect meaning. As an example we use UID message to initiate the Leader Election process, LEADER message to notify about the elected leader and so on. The messaging protocol for each message is defined as below.

MESSAGEId Separator |UID| Separator IP Separator Status Separator

MESSAGEId Separator IP Separator

MESSAGEId Separator IP Separator

Figure : Java RMI message protocol

We have implemented a message decoder for above messages as well. It decodes messages and identifies the messages and performs certain tasks according to the message type. So we have used Event Driven Architecture to do these tasks. We have grouped the things to do when a certain messages received and defined them to happen when a certain event is fired. Message decoder is responsible of firing the events and it happens according to the received event.

|  |
| --- |
| public boolean decodeMessage(final String message){ |
|  |
| //0=MessageCode |
| String[] strItems = message.split(Character.toString(MessageMeta.SEPARATOR)); |
| String messageType=MessageGenerator.getMessage(Integer.parseInt(strItems[0])); |
| System.out.println("Decoding the received message "+messageType); |
|  |
| //Starting the decoding process |
| if (Integer.parseInt(strItems[0]) == MessageMeta.LEADER) { |
| //LEADER message to inform about the new Leader |
| try { |
| if (strItems[1].equalsIgnoreCase(InetAddress.getLocalHost() |
| .getHostAddress())) { |
| //This is the leader |
| System.out.println("Localhost is the Leader"); |
| eventHandler.fireEpzillaEvent(new PulseIntervalTimeoutEvent()); |
| return true; |
| }else{ |
| //This is not the leader |
| Epzilla.setClusterLeader(strItems[1]); |
| Epzilla.setStatus(Status.NON\_LEADER.name()); |
| Epzilla.setLeaderElectionRunning(false); |
| eventHandler.fireEpzillaEvent(new ProcessStatusChangedEvent()); |
| if(Epzilla.getComponentType().equalsIgnoreCase(Component.NODE.name())){ |
| NodeClientManager.setClusterLeader(strItems[1]); |
| } |
|  |
| //Start Threading for sending data |
| Thread forwarder=new Thread(new Runnable() { |
| public void run() { |
| RmiMessageClient.forwardLeaderElectedMessage(getNextHopToCommunicate(), message); |
| } |
| }); |
| forwarder.start(); |
|  |
| Thread executor=new Thread(new Runnable() { |
| public void run() { |
| RmiMessageClient.registerListenerWithLeader(Epzilla.getClusterLeader(),  new EpZillaListener()); |
| } |
| }); |
| executor.start(); |
|  |
| eventHandler.fireEpzillaEvent(new PulseReceivedEvent(strItems[1])); |
| return true; |
| } |

Figure : Code segment for RMI Message Decoder

Above code block is from message decoder and it explains what to do when the LEADER message is received to the system.

|  |
| --- |
| public boolean fireEpzillaEvent(final IEpzillaEvent event){ |
| if(event instanceof ProcessStatusChangedEvent){ |
| System.out.println("Process status changed to:"+Epzilla.getStatus()); |
| Epzilla.resetTimerQueue(); |
| //Added later |
| EpzillaLeaderPubSub.resetPubSub(); |
| }else if(event instanceof PulseIntervalTimeoutEvent){ |
| if(Epzilla.getComponentType().equalsIgnoreCase(Component.NODE.name())){ |
| if(Epzilla.getStatus().equalsIgnoreCase(Status.LEADER.name())){ |
| Iterator<IEpzillaEventListner> iterator=EpzillaLeaderPubSub.getClientListenerList().iterator(); |
| while (iterator.hasNext()) { |
| final IEpzillaEventListner iEpzillaEventListner = (IEpzillaEventListner) iterator |
| .next(); |
| Thread sender=new Thread(new Runnable() { |
| public void run() { |
| RmiMessageClient.sendPulse(iEpzillaEventListner.getData()); |
| } |
| }); |
| sender.start(); |
| } |
| Epzilla.resetTimerQueue(); |
| Epzilla.addToTimerQueue(new PulseIntervalTimeoutEvent()); |
| } |
| } |
| }else if(event instanceof PulseNotReceivedTimeoutEvent){ |
| if(Epzilla.getStatus().equalsIgnoreCase(Status.NON\_LEADER.name())){ |
| String serverStatus=RmiMessageClient.getStateFromRemote(Epzilla.getClusterLeader()); |
| if(serverStatus!=null){ |
| if(serverStatus.equalsIgnoreCase(Status.LEADER.name())){ |
| Thread committer=new Thread(new Runnable() { |
| public void run() { |
| RmiMessageClient.registerListenerWithLeader(Epzilla.getClusterLeader(), new EpZillaListener()); |
| RmiMessageClient.sendPing(Epzilla.getClusterLeader()); |
| } |
| }); |
| committer.start(); |
| } |

Figure : Event Driven Architecture Implementation

This code block if from the fireEpzillaEvent() method. It explains what to do when the “ProcessStatusChangedEvent” event is fired and so on.

As we explained earlier the LCR algorithm implementation is very much easy task and less complex thing. Here is the code block of the LCR algorithm implementation.

|  |
| --- |
| public class LCRAlgoImpl { |
| // This is the LCR logic |
| public String runAlgorithm(String message) { |
| String[] strArray = message.split(Character.toString(MessageMeta.SEPARATOR)); |
| if (strArray != null && Integer.parseInt(strArray[1]) >= 1) { |
| int receivedUid = Integer.parseInt(strArray[1]); |
| if (receivedUid == Epzilla.getUID()) { |
| return Status.LEADER.name(); |
| } else if (receivedUid < Epzilla.getUID()) { |
| return Status.NON\_LEADER.name(); |
| } else if (receivedUid > Epzilla.getUID()) { |
| return Status.UNKNOWN.name(); |
| } |
| } |
| return null; |
| } |
| } |

Figure : Code Segment of LCR Algorithm Logic Implementation

This logic is the implementation of logic given in the LCR algorithm specification. But when it comes to the real implementation there are few more things to do. Here, some of the tasks to be done after running the algorithms.

|  |
| --- |
| //RUN LCR ALGO |
| String result=lcrAlgorithm.runAlgorithm(message); |
| //Only 3 outcomes. 1=LEADER, if UID is same. 2=NON\_LEADER, if received UID is small. 3=UNKNOWN, if received UID is large. |
| if(result.equalsIgnoreCase(Status.LEADER.name())){ |
| //Received UID is this node's UID |
| Epzilla.setClusterLeader(strItems[1]); |
| Epzilla.setStatus(Status.LEADER.name()); |
| Epzilla.setLeaderElectionRunning(false); |
| eventHandler.fireEpzillaEvent(new ProcessStatusChangedEvent()); |
| if(Epzilla.getComponentType().equalsIgnoreCase(Component.NODE.name())){ |
| NodeClientManager.setClusterLeader(strItems[1]); |
| } |
| EpzillaLeaderPubSub.initializePubSub(); |
| //Starting Sender Thread |
| Thread sender=new Thread(new Runnable() { |
| public void run() { |
| RmiMessageClient.sendLeaderElectedMessage(getNextHopToCommunicate()); |
| } |
| }); |
| sender.start(); |
| }else if(result.equalsIgnoreCase(Status.NON\_LEADER.name())){ |
| //Received UID is smaller than this UID. |
| Thread forwarder=new Thread(new Runnable() { |
| public void run() { |
| RmiMessageClient.forwardReceivedUidMessage(getNextHopToCommunicate(), message); |
| } |
| }); |
| forwarder.start(); |
| } |

Figure : State Transitions and message passing after executing the LCR Logic.

|  |
| --- |
| else{ |
| //Received UID is larger than this UID. |
| Thread sender=new Thread(new Runnable() { |
| public void run() { |
| RmiMessageClient.sendUidMessage(getNextHopToCommunicate()); |
| } |
| }); |
| sender.start(); |
| } |

Figure 36: State Transitions and message passing after executing the LCR Logic (Cont.)

Below is the sequence of tasks to be done at start up for a client node.

|  |
| --- |
| //Starting Dynamic Discovery |
| final String comType=Epzilla.getComponentType(); |
| if(comType.equalsIgnoreCase(Component.NODE.name())){ |
| @SuppressWarnings("unused") |
| NodeClientManager nodeClientMgr=new NodeClientManager(); |
|  |
| }else if(comType.equalsIgnoreCase(Component.DISPATCHER.name())){ |
| @SuppressWarnings("unused") |
| DispatcherClientManager dispClientMgr=new DispatcherClientManager(); |
| } |
|  |
| //Waiting till the dynamic Discovery discovers the other nodes,Leader and dispatchers. |
| try { |
| Thread.sleep(30000); |
| } catch (InterruptedException e) { |
| e.printStackTrace(); |
| } |
|  |
| //Check whether a leader is there. |
| if(comType.equalsIgnoreCase(Component.NODE.name())){ |
| final String clusterLeader=NodeClientManager.getClusterLeader(); |
| if(clusterLeader!=null){ |
| //No LE required. Leader Exist. Join the STM. Set the Epzilla variables. |
| Epzilla.setClusterLeader(clusterLeader); |
| Epzilla.setStatus(Status.NON\_LEADER.name()); |
| Epzilla.setLeaderElectionRunning(false); |
| eventHandler.fireEpzillaEvent(new ProcessStatusChangedEvent()); |
|  |
| Thread registrar=new Thread(new Runnable() { |
| public void run() { |
| RmiMessageClient.registerListenerWithLeader(clusterLeader, new EpZillaListener()); |
| } |
| }); |
| registrar.start(); |
| eventHandler.fireEpzillaEvent(new PulseReceivedEvent(clusterLeader)); |
| } |

Figure : Leader Election initiation process

|  |
| --- |
| else{ |
| System.out.println("No Leader Exist."); |
| boolean isDefaultLeaderNode=Epzilla.isDefaultLeader(); |
|  |
| if(isDefaultLeaderNode){ |
| //There is no other leader present and This is the default leader. |
| Epzilla.setLeaderElectionRunning(true); |
| Epzilla.setStatus(Status.UNKNOWN.name()); |
| eventHandler.fireEpzillaEvent(new ProcessStatusChangedEvent()); |
| final String nextNode=NodeClientManager.getNextNode(); |
|  |
| Thread starter=new Thread(new Runnable() { |
| public void run() { |
| RmiMessageClient.sendUidMessage(nextNode); |
| } |
| }); |
| starter.start(); |
|  |
| }else{//NOT\_DEFAULT\_NODE |
| //Node type, No leader present, not the default leader |
| final String defaultLeaderNode=Epzilla.getDefaultLeader(); |
| String defaultLeaderNodeStatus=null; |
| @SuppressWarnings("unused") |
| boolean defaultLeaderRunningLE=true; |
|  |
| //No choice but to send message without threading. :( |
| defaultLeaderNodeStatus=RmiMessageClient.getStateFromRemote(defaultLeaderNode); |
| //If this is the leader, it should be already discovered by the DD. |

Figure : Leader Election Initiation process

<ip uid="1" default="true">192.168.1.25</ip>

Figure : Default Leader Definition

Hence it is obvious that there are lots of tasks to be done in implementing the leader election and we have done that task very successfully.

## 5.10 Load Balancing

**5.10.1 Name Server load balancing**

In this approach we use a combination of both round robin and weighted load balancing mechanisms. Details of this mechanism will be discussed with the functionality of Name Server.

**5.10.2 Node Cluster dynamic load balancing**

The node clusters of epzilla are designed to be dynamically load balancing. Each node of a given cluster does a self evaluation of its operational load periodically. The throughput of each node is affected by its current load, thus guaranteeing that no node is overloaded is essential in maintaining the overall throughput of the system. Since the actual event rate changes with time, the load on the nodes of each cluster would vary with time as well. The best way to cope with the changing rates is to expand and contract the clusters with the event load.

In order for the clusters to expand and to contract with the event rate, each cluster needs to evaluate its own load level. This is achieved through measuring the performance of each node in the cluster and by using that information to generate a single performance index that represents the total load of the cluster. The CPU usage and the memory consumption of each node is measured locally. This information is periodically added to the software transactional memory of the node cluster. The current leader of the node cluster periodically evaluates all the performance details of each of the nodes and uses their average to generate a performance index for the cluster. The index is an integer value between 0 and 9.

The index is periodically sent to the main dispatcher to alter its trigger distribution algorithm and if the index is found to be too high, the cluster leader sends a RMI message to one of the idle nodes and makes it join the cluster. If the cluster leader finds that the performance index is small it removes a node from the cluster and returns it to an idle state.

## 

## 5.11 Remote Method Invocation

Java RMI is used as the main middleware of the system. Since this is a distributed application which runs on many hosts, we need to ensure that there is a proper mechanism to transfer all kinds of messages between the major processing components. Following Figure shows high level architecture of how we use the RMI to interconnect major components in our system.

We chose RMI in our project due to two main reasons.

* Entire application is written in java
* RMI is simple to use

Here we used properly defined interfaces in each component. So outsiders who communicate with each component do not need to know what is in their in the implementation but they only need to know how to use the services.

In our implementation we always try to minimize RMI look up operations as much as possible to reduce network traffic. Instead we use Hash tables to store the IP address and the remote object for particular interface. This is important in a scenario like, routing events to all the Leader Nodes in the system. It is ineffective to perform look up operation for arrival of each event. So in order to overcome these issues we use the best effective methods in most cases.

## I:\Final report\rmi.jpg

Figure : RMI Messaging Overview

## 5.12 Log based recovery

Log based recovery is another approach used to provide additional fault tolerance facility to Dispatcher of the system. Here we create checkpoint for each *trigger* assigned to the Cluster Node. These check-pointed details are important in following scenarios.

* Recover from a total failure occurred in Dispatchers - Here we can’t ask the client to send the triggers back. Then this mechanism is used to recover all the triggers by replay the logs.
* Recover from a total failure of Node Cluster – Here the Node Cluster can ask the Dispatcher to send the triggers back, by providing its *Cluster ID.*

To provide support the mentioned two scenarios, we implement customized algorithm to read log file and retrieve the *triggers*. Following figure shows a code segment of the algorithm developed to read the log file. *File name* and the *ClusterID* are the required parameters for this method. Finally it returns the triggers as a List.

public static List<String> readFile(File file, String strReq) {

long start = System.currentTimeMillis();

try {

scanner = new Scanner(file);

while (scanner.hasNextLine()) {

st1 = scanner.nextLine();

m1 = p1.matcher(st1);

m2 = p2.matcher(st1);

if (m1.find()) {

StringTokenizer st = new StringTokenizer(st1);

strmatch = st.nextToken();

if (strmatch.equals(strReq)) {

while (scanner.hasNextLine()) {

st2 = scanner.nextLine();

m2 = p2.matcher(st2);

m1 = p1.matcher(st2);

if (m2.find()) {

break;

} else if (m1.find()) {

break;

} else

recoverArr.add(st2);

}

}

}

}

scanner.close();

long end = System.currentTimeMillis();

System.out.println("Time: " + (end - start));

return recoverArr;

}

Figure : Log Based Recovery algorithm

Here we create checkpoint for *triggers* in the following format. Here header tag *CID0* represent the *ClusterID* , next line is the *trigger* and following the *trigger* is *</commit>* tag.

Figure : Checkpoint data format

CID0 Checkpoint

*"select CarDetails.CarModel where CarDetails.Year=1980 output as Details"*

</commit>

We use Regular expressions to identify the tags of the Log file. One of the regular expressions used is *("^[CID0-9]+ (.{10})$")*. This is to identify the “CID0 Checkpoint” tag of the above mentioned format.

# Chapter 6 – Tests, Results and Analysis

## 6.1 Introduction

In this chapter we will focus on describing the several of tests performed and their results and analysis. For the testing of project epZilla, we arranged different kinds of tests. This included various unit tests, performance tests, integration tests, and other tests were added as the project developed.

We use Apache Maven 2 build system in our project. This helps us to automated the build and also in the carrying out various tests. In the testing system our main focus is to identify the bugs of the system at initial point and fixed them before project grow up.

We perform different unit tests for the different components and some of the algorithms implemented. Out of several tests, we selected most critical test cases to discuss in this analysis.

We indicate the testing environment for each test case in the relevant place.

## 6.2 Unit Tests and Results

|  |  |
| --- | --- |
| Name | Multicast message Test |
| Description | This will test multicasting of the messages in Dynamic Discovery |
| Pre conditions | Connectivity to a network with a router. |
| Post conditions | Dynamic Discovery Component should keep listening for the incoming Multicast message. |
| Expected results | Get list of Dispatchers and list of Nodes |
| Results obtained | Discovery component returned the list of dispatchers and list of nodes. |
| Comments | Dynamic Discovery component works as expected. |

Table : Multicast Message Test

|  |  |
| --- | --- |
| Name | Leader Election Verification test |
| Description | This will test leader election algorithm and check whether the leaders are elected in an acceptable time frame |
| Pre conditions | Unique UID and Unique Cluster Id should be provided; Java RMI Registry should be running. |
| Post conditions | Leader Election component should keep running and check for the aliveness of the elected leader. |
| Expected results | Only one process becomes the leader and the leader should have the highest priority UID. |
| Results obtained | Process with the lowest value for UID (Highest Preference) became the leader of the system. |
| Comments | Leader election component works correctly with dynamic discovery component. |

Table : Leader Election verification Test

## 6.3 Integration Tests and Results

|  |  |
| --- | --- |
| Name | Trigger Distribution Test |
| Description | This will test Trigger distribution process of the system. Starting from the client side and add Triggers to the Dispatchers STM. |
| Pre conditions | Name Server and Dispatcher Service up and running. |
| Post conditions | Dispatcher service working with Triggers added to the STM of Dispatcher |
| Expected results | Client will receive a notification message which indicate Trigger stream is added to the Dispatcher |
| Results obtained | Client receives a message which indicates Trigger stream added successfully to the Dispatcher. |
| Comments | Trigger distribution happens successfully and expected result is obtained. |

Table : Trigger Distribution Test

|  |  |
| --- | --- |
| Name | Event Distribution Test |
| Description | This will test the event sending initialization process in the Client side. And when Dispatcher receives the Events and all the Events are Dispatch to all the Leader Nodes. Measure the time taken to send Event flow rate from Client side to the processing Nodes. |
| Pre conditions | Name Server, Dispatcher Service and Node leaders are up and running. |
| Post conditions | The successful event distribution process will result in arrival of notification from the Dispatcher |
| Expected results | Satisfying the requirements mentioned, successful Event distribution process and receipt of notifications from the Dispatcher |
| Results obtained | The time receipt of notification is normal at the given level of concurrency, and the operation succeeds satisfying the stated requirements. |
| Comments | Events are distributed to the processing Nodes in an acceptable flow. This can be considered as positive outcome. |

Table : Event Distribution Test

## 6.4 Performance Tests and Results

### 6.4.1 XSTM performance results

During our research on the possibility of using a software transactional memory to synchronize nodes we had to do a thorough performance evaluation on the possible solutions. Hence we did a network usage test and a throughput test on XSTM. The tests were done for synchronizing objects on two machines. The size of the object being shared on the STM was changed by changing the length of a string. In Java, the memory usage of a string can be calculated with the following equation

***String memory usage (bytes) = 8 \* (int) ((((no chars) \* 2) + 45) / 8)***

The in memory size of a string can be determined by the number of characters it has. Objects of various sizes were created using the above equation. We use following machines in the test environment.

* Intel(R) Pentium IV, 3Ghz with 1GB of Ram

And a

* Intel(R) Core Duo , 1.68Ghz with 2GB of Ram

The two machines were connected through a Local Area Network and the times and network usages were measured for sharing 10,000 objects via transitions between the two machines. Following are the results of the transaction rate test, it shows how the transaction rate changes with the

|  |  |
| --- | --- |
| Object Size | Transactions/sec |
| 512b | 52.844 |
| 1Kb | 52.319 |
| 2Kb | 1553.277 |
| 3Kb | 1188.636 |
| 5kb | 869.489 |
| 10Kb | 733.352 |
| 20Kb | 443.931 |
| 30Kb | 276.342 |
| 50Kb | 204.302 |
| 70Kb | 153.626 |

Table : Transation Rates Vs Object size

Figure : Transaction rate Vs Object size graph

Following are the results for the network usage test, it shows how the amount of network traffic change with the size of the objects being synchronized.

|  |  |
| --- | --- |
| Object Size | Network usage |
| 512b | 0.29% |
| 1Kb | 0.50% |
| 2Kb | 29% |
| 3Kb | 33.38% |
| 5kb | 49.90% |
| 10Kb | 62.20% |
| 20Kb | 75.20% |
| 30Kb | 82.13% |
| 50Kb | 86.56% |
| 70Kb | 89.02% |

Table : Network Usage Vs Object Size

Figure : Network usage Vs Object size graph

The results obtained in the tests were extremely useful in our design of the architecture. It showed that even for small object sizes, the maximum transaction rate possible was 52 Transactions/Sec which is extremely low compared to the event processing throughput that we were hoping to get. Hence we decided to control the usage of the STM and to only use it to synchronize the most critical data which is required to achieve fault tolerance. Hence the architecture was optimized in a way to maximize the throughput while retaining the maximum possible amount of fault tolerance.

The network usage of the STM was also not as critical as its slow object transaction rate. It did not affect our design as much. Since we were anyway optimizing the usage of the STM, the network usage factor was under control as well.

### 6.4.2 Dispatcher Event Distribution results

Here we will discuss the performance results obtained in the Event Dispatching process. Following is a description of the process carried out during the performance testing. Process initialize by calling to the event, trigger generator. There are separate threads to send generated Events and Triggers to the Dispatcher. When the Dispatcher receive the Triggers it does the dependency analysis, put the Triggers to the STM. And the Events received from the client are Dispatch to the available Leader Nodes registered in the Dispatcher. The test is done to measure the time taken to send Events to the Leader Nodes from the Client side. The entire message passing is done through RMI and we send Events as a byte stream while Triggers as a Lists. All the times are taken from the starting of first system call. Here it is call the Event sending method in the Client application.

Following are the instruments we use in the testing process.

|  |  |
| --- | --- |
| Processor | Intel Pentium IV, 3 GHz, |
| RAM | 2 GB |
| Operating system | Windows XP |
| Build system | Apache Maven 2 |

Table : The Tesing Environment

We use one machine as the Name Server, one as client application, two as Dispatchers, and two machines as Node Clusters. We used a LAN network switch to interconnect them.

**Test 1:**

Following diagrams shows the results obtained in the Event Dispatch tests. Here we send 500 Triggers for each 10 seconds while sending Events continuously to the Dispatcher.

|  |  |
| --- | --- |
| Time (s) | Event Dispatch rate (Event/sec) |
| 0 | 1 |
| 10 | 2505 |
| 20 | 1960 |
| 30 | 1400 |
| 40 | 900 |
| 50 | 750 |
| 60 | 650 |
| 70 | 575 |
| 80 | 500 |
| 90 | 550 |
| 100 | 400 |
| 110 | 350 |
| 120 | 375 |

Table : Event Dispatch rates for simultaneous process

Figure : Event Dispatching (Simultaneously with Triggers)

**Analysis:**

We can observe high Event Dispatch rate initially, but the Event Dispatching rate drops with the time and at the later stage it comes to constant rate of 40 Events/sec. Here we see a drop of Event Dispatch rate because we add Events and Triggers simultaneously into the Dispatchers and because of this STM tractions happen frequently. So we can say Event Dispatch rate reduce due to the STM transactions and with the time Event Dispatch rate comes to constant.

**Test 2:**

In the following diagrams it shows the results obtained for test mentioned below. Here we initially sent 20,000 Triggers to the Dispatcher after Event sending process started. We maintained constant Event flow in the system and we didn’t add Triggers to the system on this period of time.

|  |  |
| --- | --- |
| Time (s) | Event Dispatch rate |
| 0 | 6474 |
| 10 | 6368 |
| 20 | 5930 |
| 30 | 6043 |
| 40 | 6050 |
| 50 | 6450 |
| 60 | 6362 |
| 70 | 6250 |
| 80 | 6330 |
| 90 | 6385 |
| 100 | 6210 |
| 110 | 6280 |
| 120 | 6310 |

Table : Event Dispatch rates

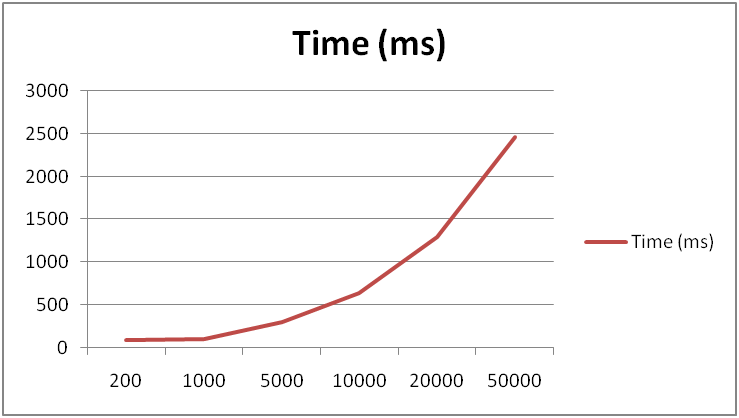
Figure : Event Dispatch Rates

**Analysis:**

Initially we get high Event Dispatch rate since Triggers sending process not initialize at that point, after we can observe sudden drop of Event Dispatch rate because we add a list of 20,000 Triggers to the Dispatchers STM. So we can observe sudden drop of performance here. In this test we add Triggers only once. Because of this in the remaining time period we observed steady Event Dispatch rate which fluctuate between, 600 to 650 Events/ sec. By comparing these results with previously obtained results we can conclude that our implementation is confirmed to work fast.

### 6.4.3 Dynamic trigger dependency analysis

The implementation of an algorithm described in a previous section was evaluated against various trigger loads for execution times.



**Figure 26 - plotting of execution time against the number of triggers fed to the analyzer.**

The test was run on a single machine with a 2.2 GHz Intel Core 2 Duo processor and a 2 GB RAM with an instance of STM running. The results imply that the implementation has below O(n2) time complexity and the implementation can successfully be used in a system to analyze query dependencies without causing significant performance overhead.

# Chapter 7 – Discussion

## 7.1 Introduction

Since the project epZilla started, our main aim was to create a distributed system architecture that would meet our expectations of scalability, performance and fault tolerance. In going through with this endeavour we faced many challenges that even threatened the mere feasibility of such a system. Most of the challenges were directly from the field of distributed computing. The merge of the two fields, distributed computing and complex event processing raised numerous challenges, a fine balance had to be found on sacrificing performance (event and trigger throughput) for the sake of achieving greater availability and reliability. The development of any distributed system has challenges on its own. Testing even the most simple of the implemented features requires multiple machines. Most of the time we had to utilize virtual machines to do testing. Integrating the separately developed features was a huge challenge as well as the debugging. Most of the debugging during the integration was done separately for the various components.

Despite all the troubles and the inherent challenges that we faced, we are extremely proud to state that we were able to develop a system that successfully satisfied our expectations and even surpassed our expectations on the actual event throughput that we were able to achieve. Following are the details of our achievements and the project outcome.

## 7.2 The Outcome

The actual outcome of the project epZilla is a distributed system architecture which acts as a framework that any organization that requires a mission critical, real-time complex event processing engine can use without worrying about the performance, reliability or scalability. The epZilla architecture can be used to process any form of events on any of the freely available event processing engines such as Esper.

Even though the system has many usages, the most critical usage and the usage with the highest market share is “Continuous Intelligence” or “Operational Responsiveness”. Operational responsiveness by complex event processing is the analysis of event data in real-time to generate immediate insight and enable instant response to changing conditions, which is a strategy that large organizations use to increase their productivity and profitability in order to cope with the continuously changing environment. The event data processing is highly critical for these organizations driven by the fact that by missing even several seconds of the event stream they can lose Millions of dollars on incorrect financial decisions or lost opportunities. Such organizations can use the epZilla system to analyze vast amounts of real-time events in order to take quick business decisions. Most of the previous practical implementations of such systems have focused on event processing through a single mainframe design, which is extremely expensive and prone to a single point of failure. EpZilla provides a cost effective, reliable and scalable alternative to mainframes.

The epZilla architecture code base APIs can be used by any organization to meet their custom requirements. Each organization can decide on the level of availability and event throughput that they require and use a desired number of machines to achieve their goals. Even though we cannot guarantee that the epZilla system implementation is 100% bug free, we can state with confidence that the system was thoroughly tested and that high quality can be assured.

## 7.3 Our achievements

During the time spent working on project epZilla we were able to achieve several technical goals which were fundamental in our design and implementation.

* Achieving fault tolerance through node replication using a Software Transactional Memory.
* The usage of Stratification techniques to make the distributed system a complete complex event processing system.
* The usage of Stratification techniques to distribute triggers among clusters based on their dependencies to increase performance.
* The usage of Dynamic Discovery to discover all the nodes related to the distributed system.
* The usage of a Leader Election algorithm to elect the leader of the available dispatchers and to elect a leader of the node cluster in order to initialize the Software Transactional Memory.
* The usage of log based recovery mechanism to recover the system from a total failure.
* The usage of dynamic load balancing to make the clusters expand and contract based on the event load.

All of the above mentioned achievements resulted in a cumulative effort in completing the project successfully.

# Chapter 8 – Future Work

Even though the implementation of epZilla as our final year project ends here, there is still a bright path ahead. There are a vast number of possible future improvements that we can do to the initial system which includes everything from small changes to the behaviour of the systems to implementing custom components to suit the exact needs of the system. The following details describe some of the possible improvements.

## 8.1 Improvements to the STM

The Software Transactional Memory framework implementation that we used for this project had a few problems from the beginning. Even though it had all the features that we were looking for it was not optimal. WE had problems with it’s high network usage and with the limited object types that were able to be synchronized among nodes. We were forced to use it because there was no other available implementation that had all the features that was required for the usage in the project, and developing such a framework from scratch was out of the question given the time duration of the project.

Hence as a future improvement of the system we suggest implementing a custom STM implementation specifically to meet the exact requirements of epZilla. This would actually improve the performance of the architecture in terms of event throughput and make the whole system more efficient.

## 8.2 Improvements to the Dynamic Discovery Mechanism

As a future improvement to project epZilla we propose some modifications to the dynamic discovery mechanism. Currently all the services in a node emit a multicast message over the network for each and every service. Hence there can be instances where these control message traffic will be critical for the performance of the system. In the future it should be reduced to a message per node where this message contains details of every service offered by corresponding node. It can further reduce to a message per cluster where cluster leader or some other node emits the multicast message about the services offered by each and every node in the cluster appended together. If we use some string compression technique, then the message length can be further shortened.

## 8.3 Improvements to the Dynamic Load Balancing

The current dynamic load balancing algorithm used in the node clusters needs to be improved to be better responsive for changing loads. The current load balancing mechanism only takes in to account the CPU usage and the Memory consumption of a given node. But it would be also useful to consider the network usage as well in generating the performance index. This would help to identify high network traffic which occurs due to larger events being processed. The event throughput can actually go down based on network saturation without affecting the CPU usage or the memory consumption.

## 8.4 Improvements to System Initialization

In the current system, initially we have to manually designate each node as a dispatcher, cluster node or an accumulator. The clusters can dynamically add or remove nodes only after the system is up and running. But as a future improvement we would use a single node to determine which of the available nodes are assigned to the specific roles based on the characteristics of the node. This would remove all manual intervention and make the system deployment much easier.

# Chapter 9 – Conclusion

Taking a look back at the very busy bringing of project epZilla we can appreciate the hard work performed by us. Without a doubt, project epZilla achieve most of the major objectives.

All the experiences gathered by us during the development process are well documented in the previous chapters. We have also continually ensured that epZilla is a product of quality, by implementing and maintaining global standards of coding as well as unparalleled software quality standards. Most part of the product at each stage has always been thoroughly tested for functionality, performance as well as the highest standards of usability.

# 

# References

[1] Akshaya Bhatia. (2008, January). “Complex Event Processing” [Online]. Available : <http://it.toolbox.com/wiki/index.php?title=Complex_Event_Processing&oldid=45313>. [Accessed: Oct. 31, 2009].

[2]. Dr. Martin Crane, Karl Podesta. (2002, November). “Distributed Systems, (A very basic introduction of real world examples)” [Online]. Available: <http://www.computing.dcu.ie/~kpodesta/distributed/>.[Accessed: Oct. 31, 2009].

[3]. Brian Selic. (2004, July). “Fault tolerance techniques for distributed systems” [Online]. Available: <http://www.ibm.com/developerworks/rational/library/114.html>. [Accessed: Oct. 31, 2009].

[4]. **Simon Brown. (2008, May). “**Scalability Principles” [Online].

Available: <http://www.infoq.com/articles/scalability-principles>. [Accessed: Oct. 31, 2009].

[5] David Luckham. “A Short History of Complex Event Processing Part 1: Beginnings”, 2007.  
  
[6] David Luckham. “A Short History of Complex Event Processing Part 2: the rise of CEP”, 2007.  
  
[7] David Luckham. “A Brief Overview of the Concepts of CEP”, 2007.

[8] Jie Wu. Distributed System Design, 1998-08-06.

[9] K. Israel, K. Mani. Fault-Tolerant Systems. San Francisco, 2007.

[10] David Powell, "Distributed Fault Tolerance: Lessons from Delta-4," IEEE Micro, vol. 14, no. 1, pp. 36-47, Feb. 1994

[11] Simon Peyton Jones, “Beautiful concurrency”, Microsoft Research, Cambridge  
May 1, 2007  
  
[12] Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy, “Composable Memory Transactions“ , Microsoft Research, Cambridge, August 18, 2006.

[13] S. Neogy, A. Sinha and P. K. Das. “Checkpoint Processing in Distributed Systems Software Using Synchronized Clocks”, Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC .01)

[14] Gengbin Zhen, et al. ” Performance Evaluation of Automatic Checkpointbased Fault Tolerance for AMPI and Charm++”.  
Available : <http://portal.acm.org/citation.cfm?id=1131322.1131340>

[15]     G.T. Lakshmanan, Y.G. Rabinovich and O. Etzion, "A Stratified Approach for Supporting High Throughput Event Processing Applications," in The 3rd ACM International Conference on Distributed Event-Based Systems, 6-9 July, 2009, Nashville.

[16]     A. Biger, O. Etzion and Y. Rabinovich, "Stratified implementation of event processing network," in The 2nd International Conference on Distributed Event-Based Systems, 1-4 July, 2008, Rome.

[17] Biger A. “Complex Event Processing Scalability by Partition”. M.Sc. Thesis, Technion. Israel Institute of Technology, 2007.

[18]   M. Michael, E. Moreira, D. Shiloach and R. Wisniewski, “Scale-up x Scale-out: A case study using Nutch/Lucene”, in Parallel and Distributed Processing Symposium, 26-30 March, 2007, Long Beach, CA

[19] Andrew S. Tanenbaum. “Distributed Operating Systems”. 2002

[20] Nancy A. Lynch. “Distributed Algorithms”. 1997

[21] Eshrat Ailiomandi, Michel J. Fischer, Nancy A. Lynch. “Efficiency of Synchronous Versus Asynchronous Distributed Systems”. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.7561&rep=rep1&type=pdf>

[22] Steve Vinoski. (2003,February). “Service Discovery 101”[Online]. Available: <http://steve.vinoski.net/pdf/IEEE-Service_Discovery_101.pdf>. [Accessed: April. 24,2010]

[23] Dan Agonistes. ( 2005, May) “Service Discovery” [Online]. Available : <http://realsoa.blogspot.com/2005/05/service-discovery.html>. [Accessed: January. 30, 2010].

[24] Philip Bishop, Nigel Warren. (2001, November). “Jini-like discovery for RMI” Available:<http://www.javaworld.com/javaworld/jw-11-2001/jw-1121-jinirmi.html?page=1>[[Accessed: March. 12, 2010].

[25] “Jini Org”, [Online]. Available: <http://www.jini.org/wiki/Main_Page>. [Accessed: March. 12, 2010].