Scalable Fault Tolerant Architecture for Complex Event Processing Systems

H.C. Randika[1], H.E.Martin[1], D.M.R.R. Sampath[1], D.S. Metihakwala[1], K. Sarveswaren[1] and M. Wijekoon[2]
[1]Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka.
[2]Creative Solutions Pte Ltd, Sri Lanka.

Abstract — **Complex Event Processing (CEP) is one of the emerging areas in computer science and it is being heavily used in real time systems during the recent times. CEP systems should be scalable and fault tolerant. The research project epZilla is to build a scalable, highly available and fault-tolerant distributed architecture for CEP systems. Software Transactional Memory, Leader Election algorithms, dynamic load balancing, dynamic service discovery, log based recovery algorithms, and Stratification are the concepts which are used to build the proposed architecture. The results of the pilot run are promising and show that the proposed architecture is scalable and fault tolerant.**

Keywords — **CEP, Distributed Systems, Dynamic Discovery, STM, Stratification**

I. Introduction

Today most of the business has increasing demand for distributed computing applications. This is mainly because of the expansion of businesses. In some of the businesses, it is required that several thousands of events be processed, which is generally not achievable using centralized systems and this motivates the usage of distributed systems in business applications.

The Complex Event Processing (CEP) paradigm is becoming increasingly popular in modern businesses. Basically it allows identifying of patterns among large amount of raw data which is processed real-time, which makes it more desirable to build CEP systems using the distributed computing paradigm. However, due to increasing loads and high availability requirements, instead of designing distributed systems with arbitrary architectures, it becomes important to take into account scalability and fault-tolerance while designing such systems. The project epZilla is a solution which provides scalable and fault tolerant distributed architecture for Complex Event Processing systems.

In this paper chapters are organized in the following mentioned order. Each of the sections is dedicated to provide the reader a good understanding of the system and all the relevant concepts used in the system will be described within the sections.

Section II, "Background", will describe the major concepts that are used in the development of the architecture and the implementation of project epZilla.

Section III, "Overview of the Components", will describe the details of the major components of the architecture.

Section IV, "Introduction to the System", will describe how the concepts that are mentioned in the "Background" are used in the actual implementation.

Section V, "Performance Results", will describe the performance details of the pilot implementation of the architecture.

Section VI, "Future Works", will describe the further development areas of the proposed architecture.

Appendix I shows the high level system architecture of project epZilla.

II. Background

This section is focused on describing major technologies and concepts used in the project.

A. Distributed Systems

A Distributed system [1] is a collection of two or more processing units. Typically distributed system uses multiple hardware units, typically hosts that are used collectively to provide a service. 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, which facilitates parallel processing.

In a multicomputer distributed system, it becomes possible to create redundant processing units with same computer programs running in different hosts. So each machine can be used to process same input in a redundant manner. In this scenario, distributed systems can work as a mirror or a replicate [2] processing unit for the original processing unit. Important properties of a distributed system include high availability, high throughput, and low latency. In order to have better performance there need to be a proper load balancing mechanism. Load balancing concepts will be discussed in a later chapter.

B. Complex Event Processing (CEP)

Complex event processing is one of the primary areas this research is based on. In simple terms, Complex event processing refers to searching of a given event streams against

a given set of queries for finding meaningful relationships among these events based on criterion defined by 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 [3].

As an example for potential CEP application, we can consider a set of transactions in a Stock Market as the set of events and we can consider a set of rules like Time difference between transactions of a same person of same listed company, number of consecutive trades of same company, fluctuations of stock value etc. In complex event processing, system searches for any matching set of transactions in the given set of transactions. If something can be derived from the incoming set of transaction details, which are events in this context, system sends a reply as an alert message or by some other means. Some other areas CEP can be used are - air traffic controllers, weather forecast etc [4-6].

C. *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 [7-8] 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 [7], 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 an 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.

D. *Software Transactional memory (STM)*

Software transactional memory [9-12] 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. The software transactional memory can synchronize any number of shared variables between any numbers of nodes thus ensuring all the nodes have identical information at all times.

E. *Remote Method Invocation (RMI)*

In today's business RMI [13-15] is mainly used as an effective approach to connect two or more Java applications run in different hosts. Typically RMI applications consist of two separate programs, a server and a client. Typically in server side, it creates some remote objects, makes references to these objects accessible, and then waits for clients to invoke methods on these objects. In a typical client side program, it obtains a remote reference to one or more remote objects on a server and then invokes methods on them. So in RMI, it provides means by which the server and the client communicate in the both direction. This is known as Call-Forward and Call-Back mechanisms [13].

Figure 1 shows a high level architecture of how Remote Method Invocation works.

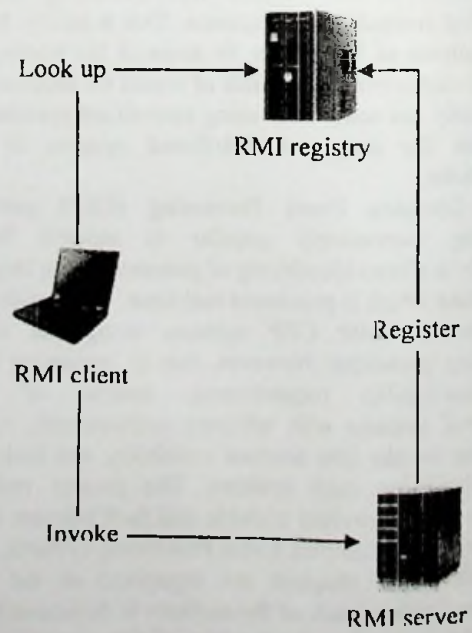


Figure 1: RMI architecture

The remote objects in RMI application can be used as normal objects and the behavior of it is described by the interface. So this can be defined as a software component that has properly defined interface and underlying operations. Services are used by the clients without knowing their underlying implementation. Ultimately clients only need to know how to access the service rather than knowing how the services work. RMI applications are also referred to as

distributed object applications where references to remote objects can be found through RMI registry. Typically what happens is server application binds a name with remote object in its RMI registry. And client looks up the remote object in the server's RMI registry to obtain the reference and then starts invoking methods on it. RMI registry holds details about server objects and provides the naming mechanism. RMI uses Java Remote Method Protocol (JRMP) on top of TCP/IP in the communication process [14].

One of the unique features of RMI is dynamic code loading. This enables dynamically extending the behavior of an application. This ability allows new types and behavior to be introduced into a remote Java virtual machine, thus dynamically extending the behavior of an application.

F. Dynamic Service Discovery

Service discovery is a concept which has evolved with the Service Oriented Architecture (SOA). Service discovery allows automatic detection of devices and services offered by these devices on a computer network [16]. This service discovery allows software to make changes to themselves while the software is 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. But in this paper we are discussing how to use dynamic discovery of Java Remote Method Invocation Services.

G. Leader Election and Election Algorithms

Leader of a distributed system handle the distribution of events, synchronization of components, etc. Since different components of the distributed system are interconnected with each other using computer networks, then we use messaging from one component to another. If the interconnections between nodes are dropped by some mean, other nodes will not be able to connect to the leader. In that case, distributed system has to elect a new leader. Algorithms we use to elect leaders in distributed systems are called as election algorithms. These algorithms are classified based on the network topology they can be applied and etc. We use low level methods such as sockets to exchange messages or higher level methods such as Java RMI. In this paper the system will use Java RMI.

III. Overview of the Components

This section aims at familiarizing the reader with the different components bundled to build up the system. System consists of four major types of components. Dispatcher, Node, Accumulator and Name Server are the main components.

A. Dispatcher

Dispatcher is one of the critical units of the system which is responsible for routing the incoming events and triggers to the Node Cluster. The systems 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). All dispatchers accept triggers from the client and add 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 number of triggers fed to the system and the interdependencies among the trigger set. Events received from the clients are not added to the STM. Instead all the events are routed to the registered Node Cluster leaders.

Figure 2 shows a high level architecture of a single Dispatcher.

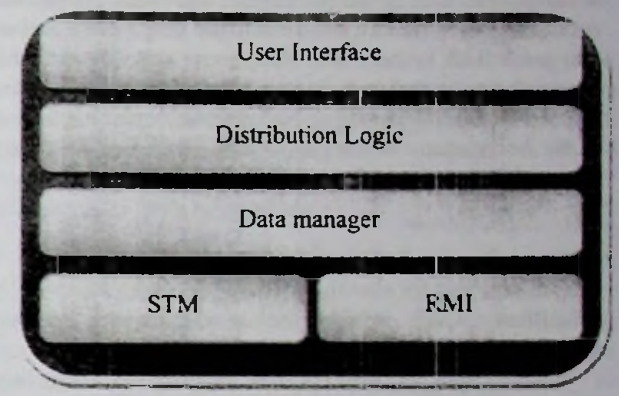


Figure 2: Dispatcher component

B. Node

The cluster nodes are the most important components of the architecture. These nodes do the actual event processing of the system. All the other nodes are performing the supporting 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.

Implementation of the cluster node facilitates the event processing engine to successfully process a stream of events. 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. Dynamic load balancing functionality will be discussed in the next section.

Figure 3 shows a high level architecture of a single Node in epZilla.

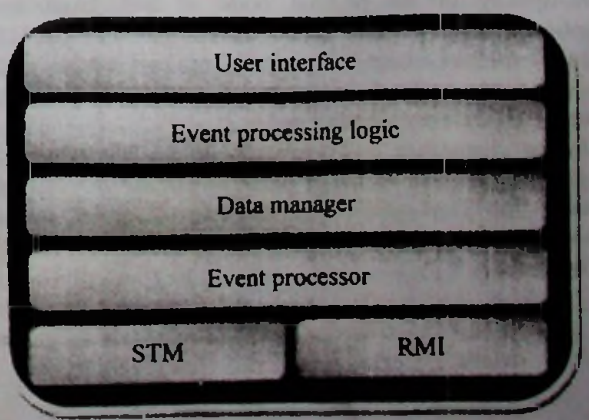


Figure 3: Node component

C. Accumulator

The systems architecture can have any number of result accumulators depending on the requirements, 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.

D. Name Server

Name Server is for keeping all the details of the Dispatchers and for allowing Dispatchers to register on it. It is a facilitator which provides lookup functionality for clients, which can be used to track dispatchers.

When a client performs the look up operation, Name Server replies the details of a Dispatcher service which has the minimum load. Load of a Dispatcher is determined based on several factors, including the number of clients connected to a particular Dispatcher and the availability of the Dispatcher. Load balancing of the Dispatchers is handled by the Name Server. This will be discussed in the next section

IV. System Functionality

This section describes the key functionality and the characteristics of the system such as Fault-tolerance, Scalability and Load balancing. Implementation details of the functions are described under the relevant topics.

A. Fault Tolerance

Fault Tolerance is considered to be one of the key characteristics of the system. The epZilla architecture provides fault tolerance by node replication. And the STM is used to synchronize the replicated nodes. Node replication is a straight forward way of implementing fault tolerance.

In the system, the critical nodes of the system are given backup nodes to replace them in the occurrence of a fault. So the backup node must be prepared to take over the functionality of the original node at any given time. Replication can be done in either active or passive mode [17].

In the solution we are using passive replication replicate the data and use software transactional memory to share the results between the replicated nodes.

In the dispatcher STM implementation it manages the scalability and fault tolerance of the dispatcher cluster 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. Implementation of Leader election and Dynamic service discovery will be discussed later in this chapter. 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.

Implementation of Accumulator and Nodes use the same concepts as mentioned above. Node cluster STM is used to synchronize the following details 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.

Since STM highly depends on the network we put extra consideration on what types of data to store on the STM. In our system, we make sure that STM is not used to contain frequently modified data. This is to make sure to reduce the number of transactions happening in STM in order to prevent unnecessary bottlenecks. In the epZilla architecture, we control the STM usage in a way which facilitates the maximum performance of the system together with the highest possible availability. Only the data crucial to the continuous operation of the system is stored in the STM

B. Scalability

The epZilla architecture is designed to achieve scalability according to the system load. Scalability is achieved in a way such that system can scale up either horizontally or vertically. The term 'cluster' is used inside the scope of the project itself rather the traditional meaning of it.

Figure 4 shows how the horizontal scaling is done in the epZilla. This is to ensure that a cluster has the capability to grow to accept the incoming load.

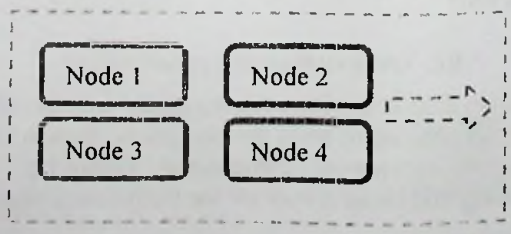


Figure 4: Horizontal Clustering (Within the same cluster)

And the Figure 5 shows vertical scalability of the architecture, which means adding clusters to the system.

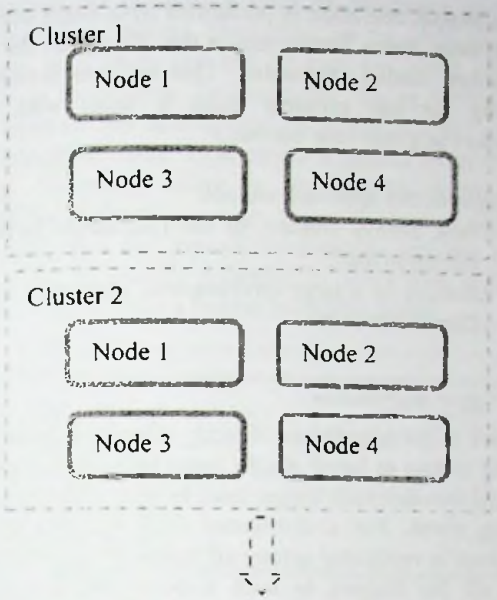


Figure 5: Vertical scaling (Add new Node Clusters)

C. *Dynamic Service Discovery*

Dynamic Service Discovery means locating network related services and devices in a computer network while system is running [16]. Jini [18] is one of the most widely used mechanisms for Dynamic Service Discovery. But in this project we do not use Jini as its mechanism is appropriate to implement a Publisher-Subscriber mechanism. Instead we use our own mechanism which is explained below. It supports both Publisher-Subscriber systems very well and locating the same components in the network.

Implementing such a mechanism for a distributed system is very useful in several aspects. Main benefit is the ability to use it for system scalability. The importance becomes significant when the system is configured for dynamic load balancing. When the rate of the inputs to the system is high, then the system's load balancing component identifies it and decides to increase the number of processing components in the system to support the increasing input rate. Therefore the dynamic load balancer can simply wake an inactive processing component using some special signal such as Magic Packet. The problem is how the existing components identify the new component and vice versa. So there should be a mechanism to identify the new components in the network by existing components and vice versa. This is where Dynamic Service Discovery comes handy.

In dynamic service discovery we keep track of the currently existing components in the system and implement a mechanism to identify the arrival of new components as well as departure of existing components. We use IP multicasting to identify the arrival of new components. Each component multicasts its capabilities and running services via multicast messages and each component have a multicast message listener and a TCP message listener. When a component is added to the system, it starts sending multicast messages and the other existing components receive the message, decode it and if the recipient is interested in about the arrival of new component to the network and if the sender and recipient belong to two component types, recipient sends a TCP message and subscribes with the new component. If the recipient and sender belong to same component type, and has the same or higher status in the distributed system, recipient does not send a TCP message but it updates its internal data structures and start tracking the status of the new component. But if the sender and recipient belong to same component type and they have different status in the distributed system and the recipient has a lower status than the sender in the distributed system, the recipient sends a TCP message to the sender and subscribe with the sender.

In this manner, we expand the system by adding new components while system is running and the new component is identified by the existing components and vice versa. Dynamic Service Discovery information is exposed via the Leader Election Component and its public API provide methods to get these information.

D. *Leader Election*

In general Distributed System is a collection of interconnected hardware components. Since there are more than one hardware unit in the system, they need to be managed in terms of communication and tasks. Therefore Distributed System requires a Leader for the system to handle these management tasks.

Leader election of distributed system was a concern for the development of the distributed systems. Therefore the computer scientists and mathematicians introduced standard Leader Election Algorithms for different types of distributed systems such as Synchronous Distributed Systems and Asynchronous Distributed Systems. These standard algorithms are proved as correct by their founders and therefore we use them without questioning their correctness.

In this project we implemented LCR Leader Election algorithm which was introduced by Le Lann, Chang and Roberts. The algorithm is named with initials letter of its founders. This algorithm is used when the network topology is a Ring and its early use was with Token Ring networks. But our distributed system is not depending on the network topology, we are executing the algorithm in a virtual ring where the sequence of the node is predefined using a configuration file and all the other basic algorithm related parameters are stated in configuration files. Distributed algorithms are executed using state transitions using the message passing. We prepared the Message protocol and state transitions.

Leader Election component uses Dynamic Service Discovery component to locate the nodes, dispatchers and accumulators. The virtual ring is created using the discovered components using the Dynamic Service Discovery.

Elected leader is reported to the system using messages and the public API of the Leader Election component provides methods to access the information about the Leader of the cluster, Discovered Nodes, Discovered Dispatchers, etc. This information is used by other components of the system for their operations. As an example, Software Transactional Memory component is initialized by the Cluster Leader. So

the STM access the Leader Election API to get the details about the Leader.

E. Dynamic Load Balancing

The dynamic load balancing [19-21] is used in following levels in the proposed architecture.

1. At dispatcher level
2. At node cluster level

a) Node Cluster dynamic load balancing

The Node Clusters of epZilla are designed in a manner such that it will balance its load dynamically. In every cluster each node 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 are 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.

Advantages dynamic load balancing approach includes:

- Performance index send to the Dispatcher is used to alter trigger distribution algorithm.
- When 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.

b) Dispatcher level load balancing

In this approach we use a combination of both round robin and weighted load balancing mechanisms. This approach becomes useful in situations where there are many clients connecting to a system which consists of multiple Dispatcher units.

In a typical scenario such as above, it is important to keep the work load of the Dispatchers at a minimum rate. When a particular client connects to the Dispatcher it will automatically update its load and sends the information to the Name Server. Name Server has the list of registered Dispatchers and it updates the load for the relevant Dispatcher.

When lookup operation is performed by a client connected to the system, Name Server replies the client with the details of minimum loaded Dispatcher. This solution seems to be promising for our problem since it deals with lot of Dispatchers in a real time manner.

Advantages of this approach include:

- Client always connect to the Dispatcher having the minimum load.
- Effective in a large environment, consist of multiple Dispatcher units.

F. Trigger distribution

The size of the trigger base directly affects the performance of a CEP system as larger trigger bases cause higher latencies due to the fact that each trigger must be executed against each incoming event. For a distributed CEP system where the trigger base is replicated among all nodes, the requirement of having all the triggers in each node causes limitation in scalability as well, where system's scalability depends on the least capable node.

To address this issue, we follow an approach where trigger base is split into several disjoint sets after analyzing the possible dependencies among all triggers in the trigger base. This approach is based on a research carried out by Lakshmanan et al. [4], in which the system is composed of several layers where outputs of each layer become inputs for the subsequent layer.

Our system consists of several clusters, each cluster having the same trigger set replicated among all nodes in the cluster. All nodes in each cluster are synchronized via a software transactional memory which is used for storing triggers and processing state.

For assigning a cluster for a specific trigger, all of its state retaining requirements are compared with all relevant existing triggers and then several disjoint sets are formed. Triggers which keep the same state are assigned to the same set. This process forms several disjoint sets which can be greater than the number of physical clusters available. Taking into consideration the number of clusters available, multiple sets are assigned to the same cluster approximating a uniform distribution. These trigger sets share the same state among all the nodes in the assigned cluster via its software transactional memory layer.

Figure 6 represents a system with 3 clusters where trigger base contains 10 distinct disjoint sets. (Each oval represents such a set of triggers where the number of queries in the set is specified inside the oval.) Each cluster is assigned multiple such trigger sets so that the trigger base is approximately uniformly distributed.

With this approach of trigger distribution, increasing the number of clusters facilitates the trigger base to be scaled up. Upon the receipt of a set of triggers, the system determines where the triggers need to be placed. This process, in the long run, causes collisions. A typical scenario will be a trigger causing two existing trigger sets to be merged. In such cases, if the two sets which are to be merged reside in two separate clusters, the system needs to resolve it by moving one trigger set so that both the trigger sets can be merged and placed in the same cluster. This process requires the system to be freeze

temporarily as processing can't be continued while trigger sets are being moved. Depending on the incoming rates of triggers, it is likely that this process can cause unnecessary delays in the system. To avoid that, we've limited the time gap between two such operations so that it can't go below a user specified value. This causes a latency to exist before a trigger becoming active in the system, however, it is acceptable as the trigger rates are extremely low compared to the event rates and smooth functionality and low latency of the system is usually preferred by users over reducing the initial latency before a trigger becoming active.

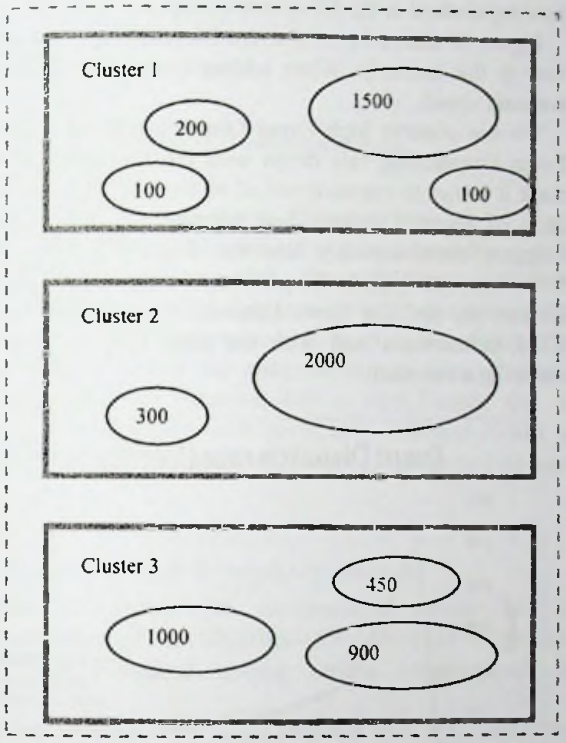


Figure 6: 3 Clusters with Disjoint Distince Trigger Sets

G. *Checkpoint based recovery*

Check-pointing [22-24] is the primary technique used in recovering data from a system crash or any failure. Project epZilla has followed this approach to keep the details of the triggers. In our implementation, a checkpoint is created for every trigger sent by the Dispatchers to the Node Clusters. Log is used in the stable storage to keep the details of the triggers. This is another approach used to provide additional fault handling capability to the Dispatcher of the system. We used regular expressions to define the tags of the checkpoint entry. For an example, the regular expression *("^[CID0-9]+ ([10])$")* is used to identify the tags in a single entry of checkpoint file. It receives the information on the *ClusterID* and the *Trigger* send to the respective cluster.

Even when a total failure occurs at the dispatcher level, it can be rolled back using checkpoint logs. It is critical to have such a mechanism since the system needs to retain all the triggers received from clients without loss. So this mechanism is used to recover all the triggers by replay logs.

To provide support for the mentioned scenarios, we implement a customized algorithm to read a log file and to retrieve the triggers.

H. *Message passing in the system*

Following Figure 7 shows a high level architecture of the epZilla [25]. It shows how the message passing is done in the project using RMI. All the shown paths are defined for the routing of the *Events* and *Triggers*. Other than this, system notifications from the Dispatcher to the Client and the *Alerts* from the *Accumulators* to the *Client* are sent using the RMI messaging.

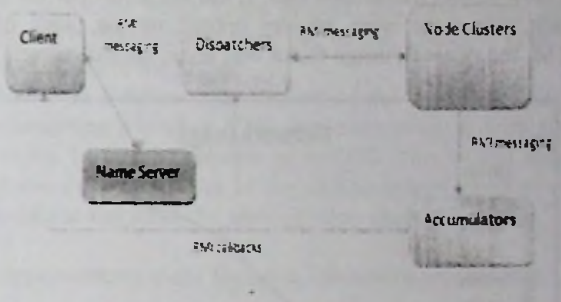


Figure 7. RMI message passing in the system

V. Performance Results

This section describes performance results obtained for epZilla during the pilot run.

A. *XSTM performance results*

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.

Figure 8 shows the results of the transaction rate test. It shows how the transaction rate changes with the time.

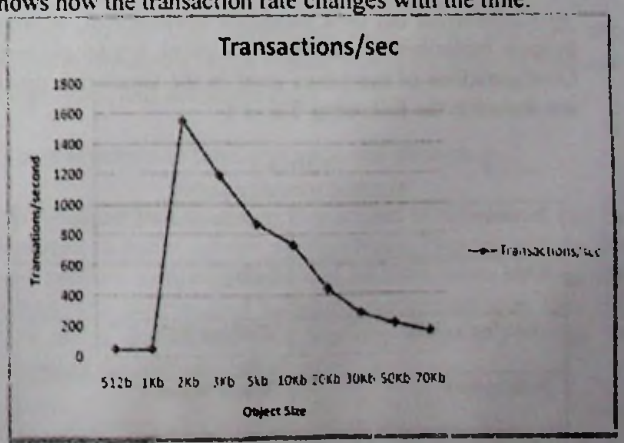


Figure 8: Transaction rate Vs Object size graph

Figure 9 shows the results for the network usage test, it shows how the amount of network traffic changes with the size of the objects being synchronized.

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.

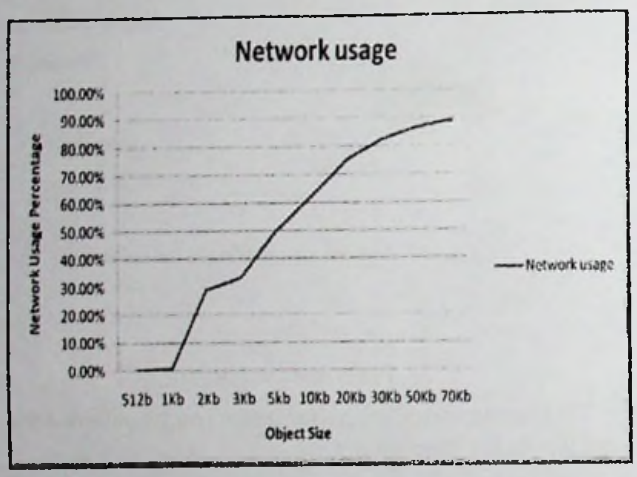


Figure 9: Network usage Vs Object size graph

B. Event/ Trigger dispatch process.

Process carried out on a simulated environment, where the process initializes by calling to the event, trigger generator. Configurations of machines used in the simulation process are shown in the following Table 1.

TABLE 1
TESTING ENVIRONMENT

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

We used Intel Core 2 Duo computers for the testing purposes in the above testing environment. We maintained separate clusters for Dispatchers and computing Nodes while allocating single node each for the Name Server and Accumulator. Test run was done for a single client in operation. All the components were interconnected using a LAN network switch.

When the Dispatcher receives the Triggers, it does the dependency analysis, puts the Triggers to the STM. And the Events received from the client are dispatched 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. All the time measures are taken from the starting of first system call. Here it is called in the Event sending method in the Client application.

Figure 10 shows how the event Dispatching rate varies with time in this scenario. When adding Events and Triggers in a constant speed.

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 STM tractions happen frequently as triggers are added. So we can say that the Event Dispatch rate reduces due to the STM transactions and with the time Event Dispatch rate comes to a constant.

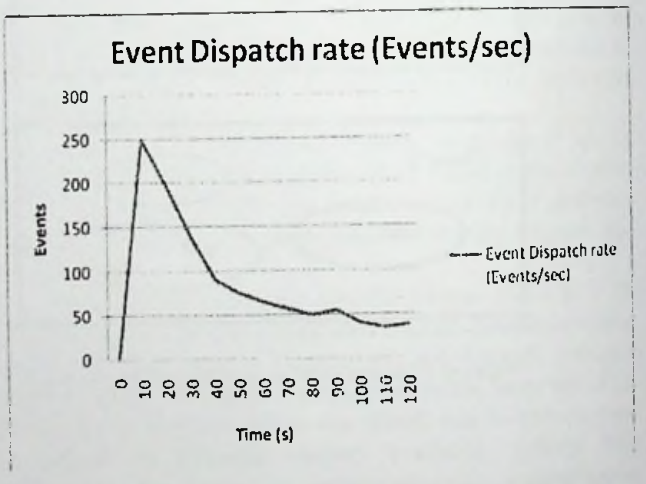


Figure 10: Event Dispatching (Simultaneously with Triggers)

Secondly, we initially sent 20,000 Triggers to the Dispatcher after Event sending process started. Then we maintained constant Event flow in the system and we didn't add Triggers to the system on this period of time.

Figure 11 shows how the Event dispatch rate varies with the time in this scenario.

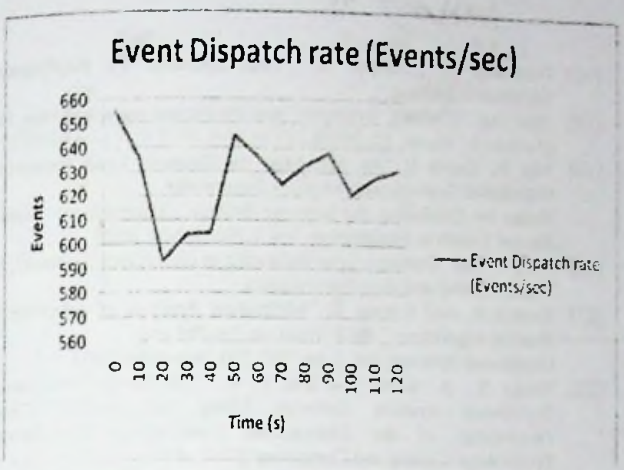


Figure 11: Event Dispatch Rates

Initially we get high Event Dispatch rate since Trigger sending process was not initialized at that point. After that, we can observe a sudden drop of the Event Dispatch rate because we added a list of 20,000 Triggers to the Dispatchers' STM. 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.

C. *Dynamic Trigger Dependency Analysis*

Figure 12 shows the performance results for the implementation of an algorithm described in a previous section was evaluated against various trigger loads for execution times.

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 $O(n^2)$ time complexity and the implementation can successfully be used in a system to analyze query dependencies without causing significant performance overhead.

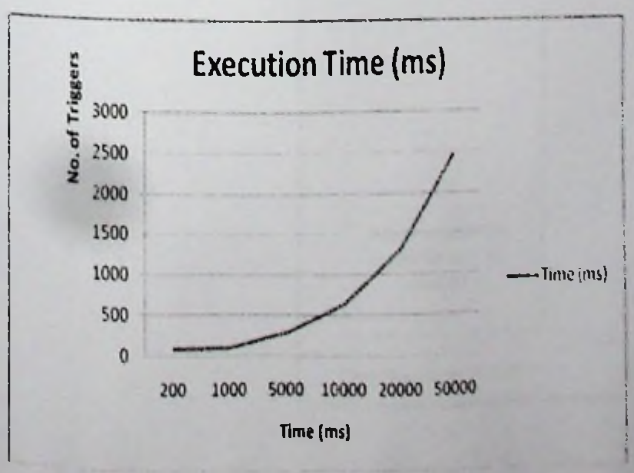


Figure 12: plotting of execution time against the number of triggers fed to the analyzer

VI. Future Work

This section is focused on describing further enhancements to the proposed architecture.

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 its 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 were 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.

2) Improvements to the Dynamic Discovery Mechanism

In the current Dynamic Service Discovery component, one component sends multicast message for each service running in the component. As an example if we consider a node, it sends a multicast message stating it is running a node service. After running the Leader Election for its cluster, if it becomes the Leader of the cluster, it starts sending a new multicast messaging saying it is running a node leader service. So now it sends 2 different multicast messages. As a future improvement to the project, we suggest to reduce the number of multicast messages by sending one message per component stating all the services it currently owns.

Another improvement is using a message compression mechanism so that the size of the message is reduced. In this manner we can optimize the bandwidth usage. But this step has to be taken with special care after considering its affect on system performance as this compression and decompression process requires some processing to be done. Therefore the system performance might reduce due to the delay in message compression and decompression.

3) Improvements to the Dynamic Load Balancing

The dynamic load balancing mechanism of the system can be improved to better handle varying loads. Factors such as the total network usage patterns can be used when scaling the system. The current load balancing mechanism only uses the CPU usage percentage and Memory usage percentage to determine the load on a single node.

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.

VII. Conclusion

A scalable and fault tolerant architecture is important for CEP systems which are typically real time distributed systems. The results of the test run of a CEP system on the proposed architecture are promising. Based on the results, we strongly believe that the proposed architecture is scalable and fault tolerant and it is suitable for deploying CEP systems.

Acknowledgment

We are grateful for insights from Dr. Shehani Weerawarna, Dr. Buddinath Jayathilake and Dr. Shantha Fernando.

References

[1] Crane M., Podesta K. (2002, November). "Distributed Systems, (A very basic introduction of real world examples)". [Online]. Available: http://www.computing.dcu.ie/~kpodesta/distributed/. [Accessed: Mar. 31, 2009].

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

[3] Bhatia A.. (2008, January). "Complex Event Processing" [Online]. Available: http://it.toolbox.com/wiki/index.php?title=Complex_Event_Processing&oldid=45313. [Accessed: Mar. 31, 2010].

[4] Chatterji G. B., Soni T., Sridar B., "Aggregate flow model for air-traffic management ". Journal of Guidance, Control, and Dynamics, 29(4):992-997, 2006.

[5] McReynolds S., "Complex Event Processing in the Real World", 2007. Available: http://www.oracle.com/us/technologies/soa/oracle-complex-event-processing-066421.pdf. [Accessed: Jun.28.2010]

[6] "Visual Analysis of Real-time Streaming data and Complex Event Processing (CEP) data", Available: http://www.panopticon.com/products/cep_complex_event_processing_real_time_visual_data_analytics.htm.[Accessed: Jun.28.2010].

[7] Lakshmanan G.T., Rabinovich Y.G. and Etzion O., "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.

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

[9] Jones S. P., "Beautiful concurrency", Microsoft Research, Cambridge, May1, 2007.

[10] Harris T., Marlow S., Jones S. M., Herlihy M., "Composable Memory Transactions", Microsoft Research, Cambridge, August 18, 2006.

[11] Herlihy M., Sun Y., "Distributed Transactional Memory for Metric-Space Networks.".

[12] Romano P., Carvalho N., Rodrigues L., "Towards Distributed Software Transactional Memory Systems.".

[13] "An Overview of RMI Applications (The Java™ Tutorials>RMI)."[Online].Available:http://java.sun.com/docs/books/tutorial/rmi/overview.html. [Accessed: April. 11, 2010].

[14] Morin S., Koren I., Krishna C. M., "JMPI: Implementing the Message Passing Standard in Java," ipdps, vol. 2, pp.0118b, International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops, 2002.

[15] Nester, C., Philippsen, M., Haumacher, B. "A More Efficient RMI for Java," JavaGrande99, pgs. 152-159.

[16] Vinoski S., (2003,February). "Service Discovery 101"[Online]. Available: http://steve.vinoski.net/pdf/IEEE-Service_Discovery_101.pdf. [Accessed: April. 24,2010]

[17] Guerraoui R., Schiper A., "Fault-Tolerance by Replication in Distributed Systems.".

[18] "Jini Org", [Online]. Available: http://www.jini.org/wiki/Main_Page [Accessed: March. 12, 2010].

[19] Jain P., Gupta D.,"An Algorithm for Dynamic Load Balancing in Distributed Systems with Multiple Supporting Nodes by Exploiting the Interrupt Service", International Journal of Recent Trends in Engineering, Vol 1, No. 1, May 2009.

[20] Philp R. Ian, "Dynamic Load Balancing in Distributed Systems", IEEE Trans. Parallel and distributed system.

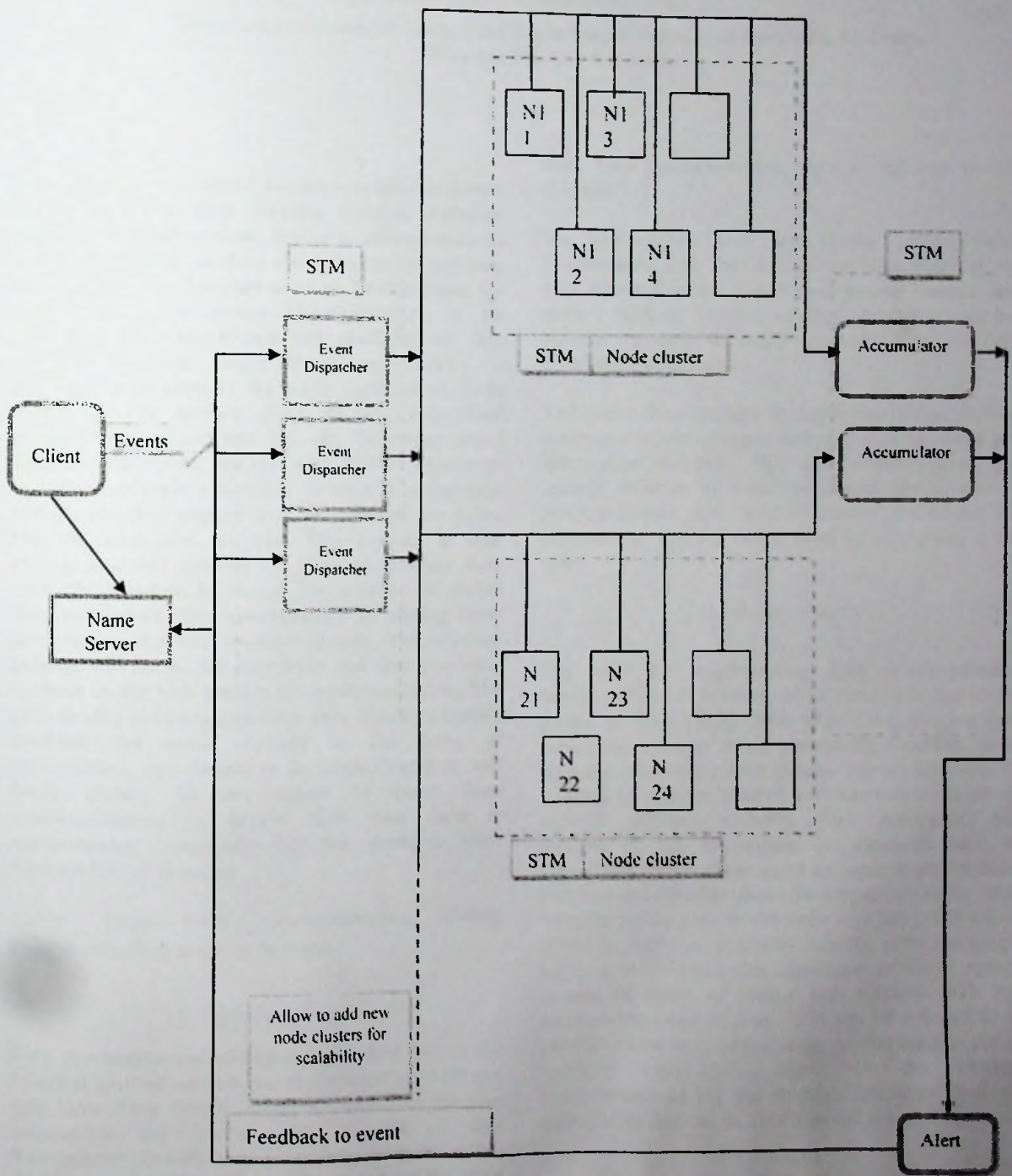
[21] Kremin 0. And Kramer J., "Methodical Analysis of Adaptive Load Sharing Algorithms," IEEE Trans. on Parallel and Distributed Systems, vol. 3, pp. 747-760, November 2005.

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

[23] Zhen G., et al. "Performance Evaluation of Automatic Checkpoint based Fault Tolerance for AMPI and Charm++".Available: http://portal.acm.org/citation.cfm?id=1131322.1131340.

[24] Wu J., "Distributed System Design", 1998-08-06

[25] Project epZilla, A scalable Fault Tolerant Architecture for Complex Event Processing Systems, http://www.epzilla.net/.


N1 1
N1 3
N1 2
N1 4
STM
Node cluster
STM
Accumulator
Accumulator
STM
Event Dispatcher
Event Dispatcher
Event Dispatcher
Client
Events
Name Server
N 21
N 23
N 22
N 24
STM
Node cluster
Allow to add new node clusters for scalability
Feedback to event
Alert