Project epZilla Evolution:

Project epZilla aims to build a distributed system to meet the problems of real-time, critical complex event processing. The essential backbone of any distributed system is the set of nodes which performs the actual processing. A set of nodes is used to distribute the processing load among them in order to gain a higher throughput. All the nodes in the system are given the same set of triggers to evaluate the events. Initially the system consists of a Dispatcher which is capable of dispatching Events/ Triggers to the processing machines. The need for a dispatcher is obvious, there should be some entity to assign events to each processing engine in order to effectively distribute the processing load. This model is a simple design but in terms of fault tolerance it has a huge drawback. The dispatcher gives a single point of failure to the system. Even if the system has 50 event processing agents, the system can go down if the event dispatcher goes down. The next evolutionary step solves this problem.

Dispatcher

Processing machine 1

Processing machine 2

Processing machine 3

Processing machine 4

The following solution to the single point of failure of the dispatcher is again a simple solution. The dispatcher has a backup node which replaces it when a failure occurs.

Dispatcher

Processing machine 1

Processing machine 2

Processing machine 3

Processing machine 4

Dispatcher Back up

The backup dispatcher waits idle till the main dispatcher goes down. This solution works fine but it is in no way the most optimal solution. For an instance, because all the processing engines have the same set of triggers each node has to evaluate each event that it gets with all the triggers. This method is highly inefficient. Following is a solution to the problem

Half of the Triggers

Half of the Triggers

Dispatcher

Processing machine 1

Processing machine 2

Processing machine 3

Processing machine 4

Dispatcher Back up

Accumulator

Accumulator Back up

This solution reduces the number of triggers that you have to evaluate the event against, by splitting the triggers among the processing engines and sending a single event to multiple processing nodes to be evaluated against their local triggers, this method can have an extremely high throughput. As an example, if there are 100 triggers in the system, and it takes T1 time for one event processing agent to process an event against all the 100 triggers. If the triggers are split between two processing agents 50 each, then it would only take an agent approximately (T1/2) time to process an event. If both agents are processing at the same time this method would increase the throughput by 100%. The set of triggers can be split between multiple agents making this job even more efficient. But in the mean time you need a special unit called a result accumulator to get those partially processed event results and turn them in to the complete result. And the accumulator needs a backup as well to avoid a single point of failure.

This is a much improved solution but it has some major problems as well. Since the backup nodes stay idle for long periods of time, they are a wastage of resources. And the backup dispatcher has to know all the trigger distributing and client details of the main dispatcher always, because the main dispatcher can go down at any time. In other words, the backup and the main dispatchers have to be synchronized at all times. And the above system is not scalable, if the event load is high the dispatcher and the accumulator can be a bottle neck. To solve all the mentioned problems above, and to make the most efficient system possible we came up with the epZilla architecture. A rough outline of it, is given below.

STM

Dispatcher

Processing machine 1

Processing machine 2

Processing machine 3

Processing machine 4

Dispatcher

Accumulator

Accumulator

...

...

...

...

...

STM

STM

Our first step was to give fault tolerance capability to the system with scalability. We needed to synchronize the available dispatcher nodes and make them scalable. After research made on this area we came up with the concept of a STM. The Software Transactional Memory synchronizes the set of dispatcher with each other, it synchronizes the set of accumulators with each other and synchronizes the node of each cluster with each other, which enabled a completely scalable system with fault tolerance.

In order to make the communication among all the machines in the system we use java RMI. And identify the each machine in the system we use Dynamic Discovery mechanism.

By effectively arranging the elements in an order and using concepts of STM, Stratification, Load Balancing, and efficient RMI messaging we came up with an architecture which use all the mentioned computing resources in an effective way to accomplish our objectives.

Here we came up with a concept call Clustering [citation] which is to make the system scale in either vertical or horizontal direction. Following figures shows how the system scales in the mentioned directions.

Figure 2: Horizontal Clustering (Within the same cluster)

Figure 3: Vertical scaling (Add new Node Clusters)

Cluster 1

Node 1

Node 2

Node 3

Node 4

Cluster 2

Node 1

Node 2

Node 3

Node 4

Cluster 1

Node 1

Node 2

Node 3

Node 4