A Scalable and Fault Tolerant Architecture for Complex Event Processing Systems

Architecture Analysis Document

Introduction

Summery

The project "epZilla" aims to create a scalable fault tolerant architecture for complex event processing systems. This distributed architecture would be a platform where multiple event processing agents work together in forming a distributed system which is scalable in the scopes of both events and triggers and the system handles the aspects of fault tolerance which is an integral part of any distributed system.

The need for a Scalable architecture for complex event processing was initiated by the fact that the number of events that modern complex event processing systems had to process differed from a few hundred per hour to several hundred thousand per minute. The number of events and triggers being processed is only determined by the use of the event processing system and the scale of the company which uses it.

This document specifies the analysis of several distributed system architectures for the purpose of gathering background knowledge for the architecture to be developed**.** The architectures analyzed in the document were chosen because of their distributed nature and association with complex event processing. All information regarding the analysis of the chosen software architectures are presented within this document.

Purpose of the Document

The main purpose of this document is to analyze various aspects of possible architectural solutions in designing and implementing a scalable and fault tolerant architecture for complex event processing systems. This document features the analysis of a set of existing and theoretical distributed architectures with the aim of identifying their properties, characteristics, values and problems with respect to their scalability and fault tolerant nature. The document contains the analysis of the solutions to the problems in each of the architectures in terms of various distributed algorithms.

The information gathered through analyzing various architectures would be essential in designing and developing an optimally fault tolerant architecture by aiding the identification of solutions to possible problems and by exposing common features of each architecture. Even though each of the architectures are analyzed, none of the architectures would be used as a whole in the architecture that

we plan to produce. The final resultant architecture that we plan to propose and implement at the conclusion of the project might include some of the features of the architectures mentioned here, features of several other architectures that are not mentioned here plus features that we ourselves have come up with.

The scope of the Document

The various architectures analyzed within this document need other external components as well to function properly. The functionality of those external components are not analyzed within the scope of this document their by restricting the focus of the analysis to only the relevant components to our study in distributed systems.

Existing Architecture at Creative Solutions

Introduction

The existing solution for the creative solutions is based on very simple and straight forward concepts and supports some degree of fault tolerance and scalability with manual intervention with the system. But this architecture has many drawbacks in securing the fault tolerance and scalability. So our aim is to create a new distributed architecture for complex event processing systems and help creative solutions company to overcome their issue by using our system.

Fault Tolerance in Existing Architecture

Existing system is consists of components such as Dispatcher, Engines, Result Accumulator which are running on hosts. Each of these components has a possibility of failures such as process crashing, host crashing or network link breakdown. So in order to provider fault tolerance, system has few mechanisms to support fault tolerance.

Dispatcher

System is usually configured with 2 dispatchers in 2 different hosts. At any given time, one dispatcher works as the Dispatcher and the other Dispatcher works as the Passive Dispatcher. When there is an Active Dispatcher, it broadcasts a unique message. Passive Dispatcher always listens to this broadcast message from the Active dispatcher and if the broadcast message is absent for a given number

broadcast iterations, Passive Dispatcher takes over the work and it start broadcasting the unique message as the Active Dispatcher.

In this architecture the Dispatcher is implemented in a manner that they do not keep any data in their hard drives related to this process or it does not load any data from hard drives other than operating system. All the relevant data are stored in the volatile memory (RAM) only. This is done to maximize the efficiency by reducing the IO operations. Hence In case of a process crash or a host crash, Dispatcher and whole system looses all the data in the process which are usually data about clients, processes, partitions, engines etc. So though the passive dispatcher takes over tasks of active dispatcher, it has to start whole work from the beginning. This is a huge drawback of this system architecture.

In a case of active dispatcher fails and then passive dispatcher takes over, then passive dispatcher also fails, then the whole system crashes since there is no dispatcher to take over the tasks. Hence in this case the degree of fault tolerance is 1. We can improve this by adding few more back up dispatchers to work as passive but when comparing this with the possibility of failures up to 2 or 3 times as once, this is extremely not feasible.

Engine

Existing architecture is designed in a manner that system does not support fault tolerance in processing engines.

System has a concept of partitions with processing engines. Usually all the partitions have similar number of engines in each of them. Dispatcher keeps track of live engines using heart beats from each and every engine in the whole system. This way if a engine fails, after a given time period the dead engine is removed from the links of Dispatcher.

When an event received to the system, dispatcher sends the event to an engine in each partition in Round Robin Algorithm. If this process/host failed after dispatcher handed over the task, there is no way that the dispatcher gets to know about it until the time out occurs. Even after time out occurs dispatcher does not send the event again to a live link. So that event does not generate any useful message at Result Accumulator since the partial message from failed engine is not received, If the process/ host failed after sending the partial message to the Accumulator, if there are no any other failure , system will create a successful message. When the next event received to the system, dispatcher will continue work with existing engines in the partition which process/host crashed.

Result Accumulator

Result Accumulator is also get fault tolerance in the same manner which is used by Dispatcher. Usually system has 3 Accumulators as Active and Passive. When Active failed, Passive takes over Active and start working.

Scalability in Existing Architecture

Scalability is another important characteristic in software systems. This existing system also provides scalability up to some extent. When considering a complex/ isolated event processing systems, there can be 2 types of scalabilities such as Scaling up/down the trigger base and scaling up/down event stream. We have to analyze the system according to each if these scenarios.

Dispatcher

In this architecture, Dispatcher is not a module which has a very processing oriented module. It only has to distribute the trigger base among the existing partitions and hand over the incoming events to an engine in each partition according to the round robin algorithm. So it down not has a huge overhead in scaling up the trigger base and/or event stream. So system does not use any mechanism for Dispatcher.

Engines

Engines are the processing units in the system. So they have a huge overhead in scaling up of triggers and/or events.

In case of a scaling up the trigger base, the system is providing the support for the additional triggers by a new partition to the system. A partition means a logical formation of engines created by the Dispatcher to manage the functionalities of the system with Scalability and Fault Tolerance. Generally all the partitions are consists with equal number of engines and all the engines in a partition have the same trigger base. So the complete trigger base is distributed among partitions by dispatcher. If we need to add new set of trigger either we have to update the trigger base of an existing partition or create a new partition and assign the new triggers to it. If the existing partitions are running with their maximum feasible number of triggers, then the system has to add a new partition in case of expanding the trigger base.

When considering the expansion of event stream, system will support scalability by a different mechanism. Since all the engines in a partition have the same set of triggers, system can add few more engines to each partition and improve the availability. Then system will support more simultaneous event processing in a partition. Hence the engines will support event processing scaling up.

Result Accumulator

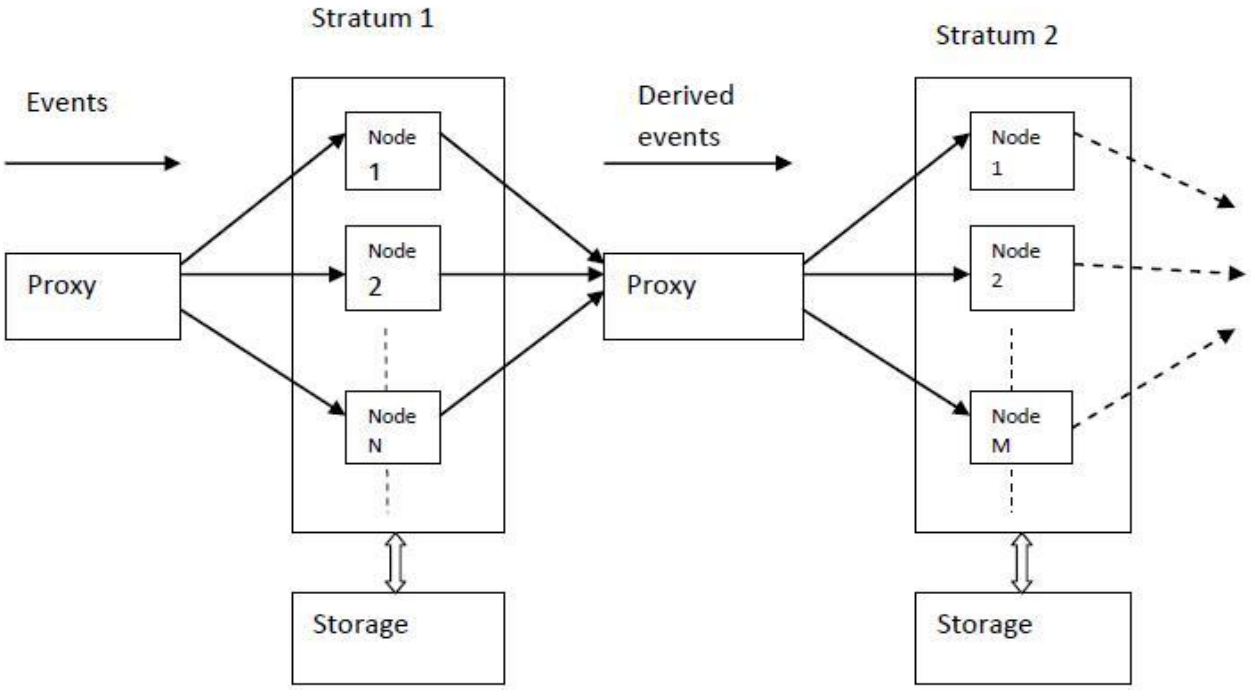
Since the result accumulator is not much processing oriented, system does not have a specific mechanism for Result Accumulators in case of scaling up.

Scalability through stratification – IBM research

Reference: http://www.slideshare.net/opher.etzion/debs-presentation-2009-july62009

Introduction

Concentrates on improving throughput in Event Processing Networks [1] through a technique called stratification. The concept is derived from earth resource engineering where 'stratum' refers to a layer. The basic idea here is to stratify (i.e. make layers) the EPN so that different event processing operations are carried out by different stratas, which collectively results in complex event processing. The main concern here is to improve the scalability in both configuration and traffic without considering the fault-tolerance.



Resource allocation and scalability

First, the solution attempts to define the event processing application as an directed event processing network dependency graph, where edges are from event sources to event targets. Manual partitioning in such application is considered to be a hectic process, so this aims to provide the specification of the application instead and then automatically define the stratas.

An abstract representation of the stratification algorithm is as follows:

1. Find sub-graphs that are independent of each other and create partitions.

2. Create a network of event processing agents [2] for each such sub-graph.

3. Push filters to the beginning of the network to filter out irrelevant events.

4. Identify sub-graphs of no connecting edges. (strict interdependence)

5. For each sub-graph, define stratum levels.

Note that this doesn't involve any observation/evaluation of available hardware. This only does a logical partitioning of the system. However, the first part of this stratification algorithm requires extensive knowledge on the domain, so in our opinion, the first part of the process, i.e. representing the entire system in a event processing network dependency graph, will be only doable by the CEP application developer and the result will be used to carry out the rest of the process.

Next step will be to assign specific hardware to each component. The strategy of performing this operation is as follows:

Profiling nodes: processing capability of each node is analyzed by applying an artificial workload.
Computing ratio of events split between stratum I.
Determine the number of nodes required for the stratum I.

Repeat for stratum i+1.

For assigning nodes to each stratum, user set percentages (of processing capability) of nodes are also considered. With this operation, the initial setup ends and the system will be ready to start complex event processing.

Load distribution

Load distribution in this system is mainly carried out by event proxies which collect statistics and maintains a time series.

The statistics collected by the event proxies are as follows:

1. Number of input events processed by execution of agents in a particular context.
2. Number of derived events produced by the execution of agents in this context.
3. Number of different agent executions evaluated in this context.
4. Total amount of latency to evaluate all agents executed in this context.

According to the researchers, the algorithm they propose for load distribution is as follows:

1. Identify most heavily loaded node in a stratum (donor node).
2. Identify a heavy context to migrate from the donor node. (Also use load correlation as a guiding factor).
3. Identify recipient node for migrated load.
4. Estimate post migration utilization of donor and recipient nodes. If post migration utilization of recipient node is unsatisfactory, go back to step 3 and identify new recipient node. If post migration utilization of donor node is unsatisfactory, go back to step 2 and identify new context to migrate.

5. Execute migration and wait for x length time interval. Go to step 1.

Note that here the term 'load' refers to the incorporated priorities and the preferences of application priorities. The term 'donor' refers to a node which will donate its load, and similarly the term 'recipient' refers to the ones which will accept the migrated load.

Post migration utilization is carried out to estimate the consequences and to avoid occurrence of unwanted consequences such as recursive migrations (due to instability of the recipient nodes).

Fault – tolerance

This architecture doesn't pay special attention for fault-tolerance and specifically aiming to improve scalability. Hence, there are many drawbacks in the system which makes it heavily vulnerable to faults.

We can analyze this architecture's fault-tolerance in terms of:

I. Node failures – entire node fails
II. Process failures – an agent fails.
III. Network link failures – communication link breaks.

Another viewpoint which can be used to analyze the fault-tolerance would be:

I. Intra-stratum failures – failures in the previous classification occurring within a single stratum – these are relatively easy to fix and doesn’t cause entire system to break.
II. Inter-strata failures – failures in the previous classification occurring between multiple strata – these can cause entire system to break down and are very hard to fix.

Node failures.

The system has a storage unit for each stratum where the results can be stored, but doesn't contain any persistent or volatile storage for storing inputs. Hence, once a node or a process failure occurs, the input events fed to it will be lost. One simple solution to this problem will be to store incoming events each stratum, but this may require a lot of storage. However, since each stratum has to temporarily store the

processed results (derived events) temporarily or permanently for processing, another simple fix would be to ask for the inputs again from the previous stratum. This should be handled by the event proxy, by probably maintaining records on which events (or ranges of events, if chunks of events are dispatched to nodes) were dispatched to which node. Once the failure occurs, the event proxy needs to fetch the results again from previous strata. This procedure will occur recursively, propagating from stratum i to stratum i-1, until one stratum is able to produce the required inputs. However, whenever this reaches the stratum 1, it will have to either fetch the inputs again from the feeder, which may not be the preferred way for a fault-tolerance system, or else, the stratum 1 can be implemented somewhat differently from other strata where it could have a cache to store all the inputs received. Once the inputs are received, they can be dispatched to the same or different nodes as just another event (depends on the context). Whenever a node fails, it results in a loss of processing power in a given stratum. So, in order to face such conditions, redundant nodes will have to be kept for each stratum to replace the failed nodes. Otherwise, this will result in an overload in the available nodes in the stratum and eventually make the stratum unstable.

To handle failures in event proxies, one simple and straightforward approach will be to keep redundant event proxies. Since the event proxy is responsible for collecting statistics, the redundant event proxies will have to be kept updated even when they are inactive. In case of an active event proxy failure, the inactive shadow proxy can be activated and if it contains the required information up to date, the system can continue operation.

Process failures

Process failures can also be addressed in the same manner mentioned under the 'node failures' section. However, since such failures don't affect other processes running in the system, we can think of a more efficient implementation for overcoming process failures. That is, for each process, i.e. event processing agent, there can be a shadow process which is running in the background while the active agent is performing without failure. Once the failure occurs, it will be up to the event proxy to activate this shadow event processing agent, and once activated, the shadow event processing agent becomes the active event processing agent. The same agent can be used to create another shadow agent just after its activated and also to kill malfunctioning processes which can remain due to the failure of the previous

agent. Until an event processing agent is active, it can feed the shadow agent too, which will make the steps mentioned in previous section to fetch inputs are no longer needed. However, this makes the agents twice more memory intensive.

The same simple approach may be used to handle process failures in event proxies too.

Network failures

Whenever a network failure occurs, there's no way of making use of the isolated nodes until the links are fixed. Whenever such scenarios occur within a stratum, the situation will be usually similar to a node failure.

Whenever a link between two strata breaks, i.e. a link between a stratum and the event proxy fails, the situation will be analogous to an event proxy node failure. So a similar approach can be used to solve these issues. However, this process can get complicated if we consider a situation where both active and redundant event proxies have to be reached through a single network link. In such cases, if that link breaks, keeping redundant proxies will also be useless, and then the entire system will go down since there is no way of communication between two adjacent strata. This is one of the hardest challenges to overcome in this architecture.

Existing distributed system architectures

Sinfonia: A New Paradigm for Building Scalable Distributed Systems

Introduction

Sinfonia[1] is a model for developing scalable distributed systems. Which doesn't require dealing with message passing protocols used in traditional distributed systems. Main part of Sinfonia is a mini-transaction process, which hide all the complexities that arise from concurrency and failures. Simply Sinfonia is a service that allows hosts to share application data in a fault-tolerant, scalable and consistent manner. Here Sinfonia provides options for developers to design and manipulate data structures to build scalable distributed systems.

Design Principles used in Sinfonia

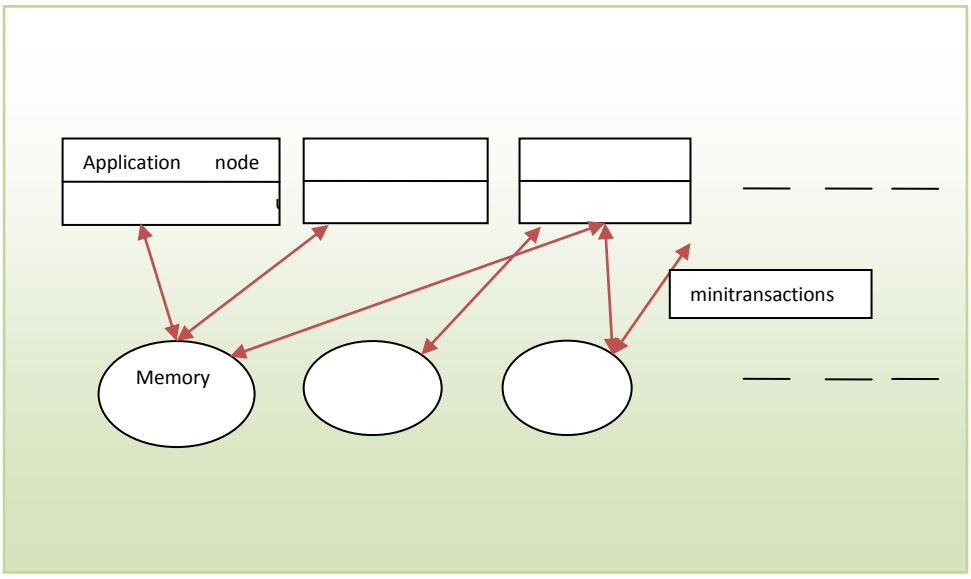
- Reduce the dependencies between the operations to obtain scalability. Here it improve the parallel execution on different processors.
- In Sinfonia it make sure that the individual components are reliable before scaling the system. Basic idea is to build a large system with many reliable components.

Basic components of Sinfonia

- Memory nodes – hold application data, either in RAM or on stable storage, according to application needs.
- User library – implement mechanisms to control data at memory nodes. Here it provides basic read/write of a byte range on a single memory node and runs on application nodes.

Application nodes and memory nodes are logically distinct but may run on same machine. Linear address space reference via (memory-node-id, address) pairs.

Following figure shows the inter connection of the mentioned components



Minitransactions

Use minitransactions to provide consistency in distributed systems.

Minitransactions guarantees :-

- Atomicity – means minitransactions executes completely or not at all.
- Consistency – means data is not corrupted.
- Isolation – means minitransactions are serializable.
- Durability – means minitransactions are not lost even if there are failures.

Minitransaction contains

- Compare items
- Read items
- Write items

Here each item specifies a memory node and an address range within that memory node. Compare and write items include data.

Operation of minitransaction :-

- check data indicated by compare items (equality comparison)
- if all match then
- retrieve data indicated by read items
- modify data indicated by write items

Minitransactions provide a base to handling distributed data. Following are the examples of minitransactions.

- Swap – here the read item returns the old value and a write item replaces it.
- Compare-and-swap – here the compare item use a constant to compare current value, and if the values equal write item replaces it.
- Atomic read of many data
- Acquire a lease – here a compare item check if a location is set to 0, and a write item sets it to the (non-zero) id of the leaseholder and another write item sets the time of lease.
- Acquire multiple leases atomically –there are multiple compare items and write items, but the procedure is same as above.
- Change data if lease is held – compare item checks that a lease is held and, write items update data.

Application uses the user library to communicate with memory nodes through RPCs. Minitransactions are implemented on top of this.

Minitransaction protocol

Use minitransactions with two phase commit. To avoid coordinator(application node) crashes traditional transactions uses three phase commit. But Sinfonia tackle this problem by blocking participant crashes. Because participants are the memory nodes which contain application data. Since the coordinator has no log it consider transaction to be committed if all memory nodes(participants) have yes vote in their log. Here is the process use for recovery from a participant(memory node) crash :-

- Block until participant recovery
- No recovery coordinator for memory node crashes
- Lose stable storage – recover from a backup
- keep stable storage – recover using redo
- Processed-point variable (check point)
- Redo-log & decided list

Following are the pseudo code segments mentioned in the research paper, which used in application node and memory nodes.

Code for coordinator ***p*****:**

To execute and commit minitransaction (*cmpitems, rditems, writems*)

tid ←new unique identifier for minitransaction

{ Phase 1 }

D ←set of memory nodes referred in *cmpitems* ∪ *rditems* ∪ *writems*

p **for** each *q* ∈ *D* **do** { pfor is a parallel for }

send (EXEC&PREPARE , *tid,D,*

πq(*cmpitems*), *πq*(*rditems*), *πq*(*writems*)) **to** *q*

{ *πq* denotes the projection to the items handled by *q* }

replies ←wait for replies from all nodes in *D*

{ Phase 2 }

if $\forall q \in D$: *replies*[*q*].*vote*=OK **then** *action* ← *true* { commit }

else *action* ← *false* { abort }

p for each $q \in D$ **do send** (COMMIT*, tid, action*) **to** *q*

return *action* { does not wait for reply of COMMIT }

Code for each participant memory node *q***:**

upon receive (EXEC&PREPARE *, tid,D, cmpitems, rditems, writems*) **from** *p* **do**

in-doubt ← *in-doubt* $\cup$ *{*(*tid, cmpitems, rditems, writems*)*}*

if *try-read-lock*(*cmpitems* $\cup$ *rditems*)=*fail or try-write-lock*(*writems*)=*fail*

then *vote* ← BAD-LOCK

else if *tid* $\in$ *forced-abort* **then** *vote* ← BAD-FORCED

{ *forced-abort* is used with recovery }

else if *cmpitems* do not match data **then** *vote* ← BAD-CMP

else *vote* ← OK

if *vote*=OK **then**

data ←read *rditems*

add (*tid,D, writems*) to *redo-log* and add *tid* to *all-log-tids*

else

data ← $\emptyset$

release locks acquired above

send-reply (*tid, vote, data*) **to** *p*

upon receive (COMMIT*, tid, action*) **from** *p* **do** { *action*: *true*=commit, *false*=abort }

(*cmpitems, rditems, writems*) ←find (*tid,* $*$*,* $*$*,* $*$) in *in-doubt*

if not found **then return** { recovery coordinator executed first }

in-doubt ← *in-doubt* − *{*(*tid, cmpitems, rditems, writems*)*}*

if *tid* $\in$ *all-log-tids* **then** *decided* ← *decided* $\cup$ *{*(*tid, action*)*}*

if *action* **then** apply *writems*

release any locks still held for *cmpitems* $\cup$ *rditems* $\cup$ *writems*

Fault tolerance in Sinfonia

Sinfonia provide fault tolerance according to the application needs. Sinfonia provide following optional protection options to the applications:-

- Mask independent failures: let say few memory nodes crashes, but Sinfonia covers the failures so that the system continue its operation without any crash.
- Preserve data on correlated failures: if there a situation where lot of memory nodes crashes within a short period without losing the stable storage, Sinfonia protect the data until enough memory nodes restarted.
- Restoring data on disasters: in a situation where memory nodes and their stable storage crash, Sinfonia recovers data using transaction ally-consistent backups.

Sinfonia uses disk images, logging, replication and backups to provide fault tolerance. Disk image use to keeps a copy of data at a memory node. A log keeps recent data updates, and the log is written synchronously to ensure data robustness. When a memory node recovers from a crash it uses recovery algorithm to replay the log. To provide high availability Sinfonia uses replicate memory nodes to take over the failed nodes. Backups made from a transaction ally consistent image of Sinfonia data.

Load balancing in Sinfonia

Sinfonia allows applications to choose where to put application data. Also Sinfonia doesn't balance load across memory nodes. But it provide per memory node information to applications. Which include bytes read and written and minitransactions executed, retried, and aborted in a various time frames(1min, 5min, 1hr,..). Applications use class identifiers to label each minitransactions and load information is provided for each class. Here the minitransactions can migrate data automatically without expose variations due to partial migration.

Our view on Sinfonia

Sinfonia works fine in independent failures of memory nodes, but it still provide overhead when restoring data from backups. Creating backups using consistent image of Sinfonia data will degrade the system performance. Also keeping disk images from memory nodes also affect the performance of the system. providing replicate memory nodes to take over failed nodes is ok rather waiting for failed node to restart. If using disk images also with replicate memory nodes, it will provide high efficiency in the process.

Since load balancing is a important factor in a distributed system, it is better if there is a mechanism to balance load among memory nodes. Using various time frames to execute, retry and abort minitransactions is good approach. But still it can be improved using better load balancing algorithm.

Possible solutions for distributed Load balancing

Following section is a description of possible solutions for distributed load balancing problem. Here we described about the existing distributed load balancing techniques. Before move into the details of algorithms it is better to start with nature of load distribution or load balancing techniques used in distributed systems.

Load distribution is the process of transferring units of work among processing elements during execution to maintain balance across processing elements. Dynamic load distribution also called load balancing, load migration or load sharing.

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

Components of dynamic load distribution[2]:

Typically there are six policies used in existing dynamic load distribution algorithms. Following is a description of them.

Initiation policy – which is to decide who should invoke load balancing activities. In a sender initiated method the heavily loaded nodes initiate the load transfer process. In a receiver initiated method the lightly loaded nodes start the load balancing process.

Transfer policy – determine whether the ode is in suitable state to participate in task transfer.

Selection policy – determine which task should be transfer

Profitability policy – it is used as an estimate of potential profit obtainable through load balancing and weighed against the load balancing overhead to determine whether or not load balancing is profitable at that time.

Location policy – determine to which node a task selected for transfer should be sent.

Information policy – triggering the collection of system state information.

Parameters used for load balancing in existing systems:

Typically there are few parameters to considered when selecting a load balancing algorithm.

System size – no of processors in the system

System load – load on processors

System traffic intensity – arrival rate of tasks to different processors

Migration threshold – load levels to trigger task migration

Task size – size of a task sufficient for migration process

Overhead costs – communication and task placement cost

Response time – turnaround time for a task

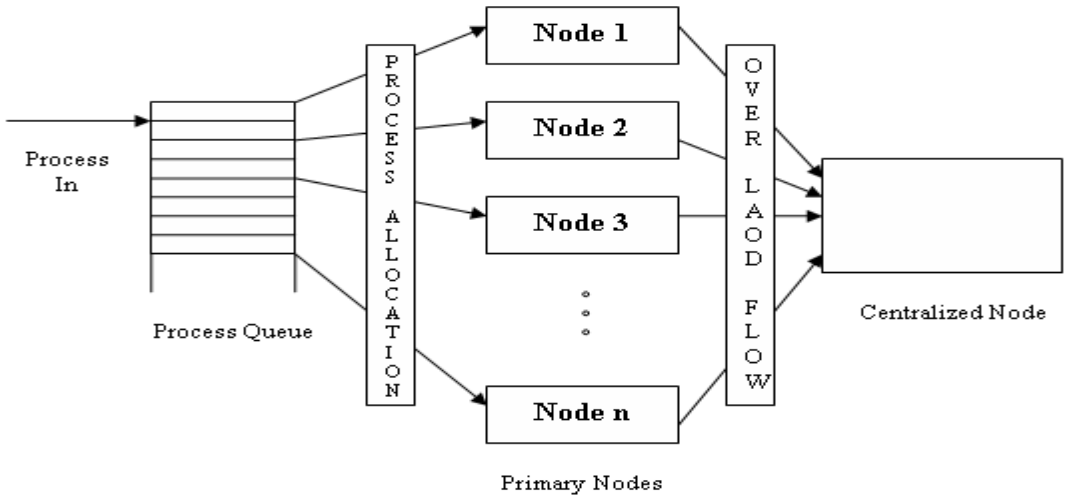
Load balancing horizon – no of neighbors to probed in determining task destination

Resource demands – demand of system resources by a task

Existing load balancing algorithms

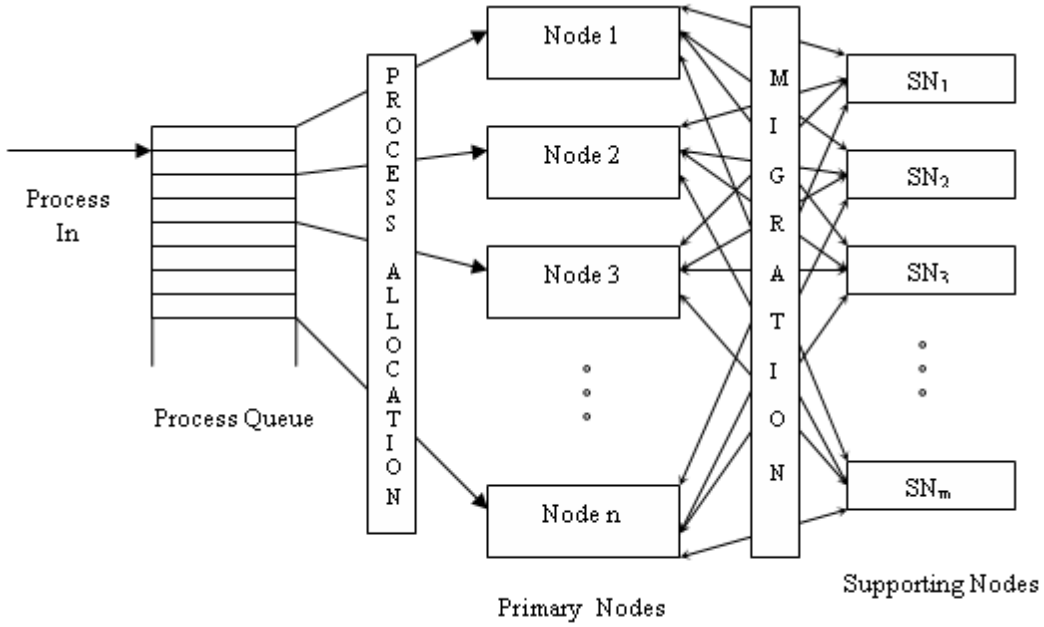
Centralized approach for load balancing part 1

In this approach[3] processes initially stored in a queue or they are allocated to nodes according their arrival. Processes migrated from heavily loaded node to light weighted node. To reduce the network traffic nodes are grouped into clusters. To increase the output of each node there is centralized node for every cluster that is to take the load from heavily loaded node if it cannot find a light weight node in the system. This centralized node is not assigned to any process initially, it is only to take overload of primary nodes. To evade any network delay traffic between primary nodes and centralized node kept in minimum value. Following figure shows how the primary nodes and centralized node arranged in a particular cluster.



Centralized approach for dynamic load balancing part 2

This is a derive approach from the above mentioned centralized load balancing approach. Here it focus on avoid speed limitation of the centralized node. Idea is to split the centralized node into small pieces called supporting nodes. These supporting nodes are not assign to any process initially, they only take the overload from the primary nodes. But take the maximum use every supporting node is given some load initially and it keep a priority queue of how processes execute in it. There is interrupt service routine to handle the interrupts coming for a particular supporting node. Here the interrupt service routine compares the priority of coming processes from primary nodes with currently executing process and switch to the most prioritize process. Each supporting node keep a priority queue where processes to be executed are sorted according to their priority. Following figure shows how the primary nodes (Node 1, Node 2……, Node n) and supporting nodes (SN_1, SN_2,……., SN_m) are organized in a single cluster.



Following is the description of the pseudo code mentioned in the paper[3]:

N_i: List of primary nodes

SN_j: List of supporting nodes

PI: Priority index maintained by supporting node

Pk: List of processes in Process Queue

i, j, k ϵ N, j<i // N← Natural Number

Initially:

1. Assuming each node is having some load,

2. Some supporting nodes may have load

N_i ← Load, S_j← Load // load defines some process is there

Procedure: Main ( )

{

I. Suppose a node N_t is heavily loaded with load ξ where 0< t< i , ξ ϵ P_k

// Two situations are possible as in step II.

II. If (Search_node)

// Search_node will find out whether a light weighted primary node available, if primary node is available it will give index of available node.

{

Available_Node ← ξ // Overload is assigned to Available node given by Seach_node ()

Load (N_t) =Load (N_t) – ξ

}

Else

{

Call Seacrh_S_node ( ) // Search_node will find out a light weighted or minimum weighted supporting //node with index as Available_S_node.

Available_s_node ← ξ

Load (N_t) =Load (N_t) – ξ

IN_S (Available_s_node, ξ)

}}

Procedure: IN_S (SN_node, ξ)

{

I. Priority is assigned to SN_node as PI

(SN_node) = t

II. If t > PI (RP) // RP is the currently process on Selected supporting node.

{

// P_List is list of pending process shorted according to priority.

P_List ← RP // RP is added to Pending List

Now RP ← ξ // ξ is now allotted to the Selected Supporting node.

}

Else

P_List ← ξ // ξ is added to Pending List

}.

}

Procedure: Search _node ()

{

I. for each Ni except node initiating the search_

node procedure

{

Check the node with minimum load

//Minimum load includes the number of

//processes as well as structure or configuration

//of node and node should be able to accept

//node.

}

II. If Desired Node available

{

return (index of available node) // index defines the property to identify the Node

}

}

Procedure: Search _S_node ()

{

I. for each SNj except node initiating the search

node procedure

{

Check the Supporting node with minimum

Load

// Minimum load includes the number of

//processes as well as structure or

//configuration of node

}

II. return (index of available Supporting node)

// index defines the property to identify the Node

}

Our view on distributed load balancing algorithms

In the centralized approach sometimes it is not necessary to keep central node to take overload from the nodes. Because if there is a situation many primary nodes try to connect the central node for balance their individual loads it will definitely degrade system performance, by causing network traffic.

But the last method is a better approach to balance the load in a distributed system. since there are lot of supporting nodes to take overload from primary nodes, also decrease network traffic since primary nodes have option to contact one from lot of supporting nodes rather than one central node. But still nodes in a particular cluster has to have lot of supporting nodes. This approach is suitable for situations where the number of computing nodes is high and to process millions of triggers in few seconds.

In the last two methods it doesn't describe how to achieve fault tolerance, in node failures and link failures. Scalability of the system can be improve to greater extent in the mentioned cases. Also the system require lot of hosts to implement the solution.

References

[1] Aguilera, k. Marcos, Arif Merchant, Mehul Shah, AlistairVeitch and Christos Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems.
Url:http://w*ww.sosp2007.org/talks/sosp064aguilera.pdf.*

[2] Lampson, B. W., "Hints for computer system design", Proc. of the 9th ACM Symp onOperating Systems Principles, 1983, 51-81

[3] Jain, Parveen and Daya Gupta. An Algorithm for Dynamic Load Balancing in Distributed Systems with Multiple Supporting Nodes by Exploiting the Interrupt Service.
Url: http://www.academypublisher.com/ijrte/vol01/no01/ ijrte0101232236.pdf