CHAPTER 5
FAULT AWARE LOAD BALANCING FOR CONTENT DELIVERY NETWORKS

This chapter proposes a fault aware load balancing algorithm for Content Delivery Networks (CDNs). CDNs are used in Cloud Model for high-speed communication. It also presents the Simulations, experimental results and comparative study with other contemporary existing algorithms. Finally the findings and its future prospects are summarized.

5.1 INTRODUCTION

With increasing use of data sharing, traffic over internet has increased tremendously. There is a need to effectively manage load over servers and maintain overall system performance with better Quality of Service (QoS). To maintain better QoS, Content Delivery Network (CDN) is used CDNs offer better network performance by optimal bandwidth utilization, improved accessibility, maintaining correctness through content replication that result in load reduction on servers[125][127][128][166][167]. The limitation of existing CDN load balancing algorithms is that it considers servers and the systems as non-faulty. This increases the probability of request being getting allocated to faulty server. With increase in number of requests, failure probability increases due to long waiting queue which increases the network load and processing time. To overcome this problem, a fault aware load balancing algorithm for CDNs is proposed that improves the QoS and reliability of the system. Effect of network failure on QoS and reliability of system is studied in presence of high request rate and network traffic. Performance of existing load balancing algorithm is investigated and compared in faulty environment. Moreover, the performance of the proposed algorithm is compared with reported techniques. The experimental results demonstrate that proposed algorithm provides better robustness and resilience to fault without affecting the QoS. Further, a dynamic fault model is proposed and implemented incorporating changing failure probability with load which gives better result as compared to static fault models.
CDN is a popular solution to balance load over a distributed system which acts as a single system for users. Different CDN based proposals are made for cloud model [125][126][127]. Few proposals are for balancing load on the basis of Cost, Response time and load on servers [128][168][169]. Energy efficient and rate of data transfer based proposals are also made [129][130]. Load balancing in CDN is a major concern [170][171]. A proposal for scalable and reliable architecture for CDN along with fault aware load balancing algorithm is made. Although many existing approaches address the issue of load balancing in CDN but they do not take into consideration failures at servers which increases with increase in load. The proposed algorithm takes into consideration both load and failure over a server and scalability of CDNs. The proposed algorithm tries to solve the problem of scalability and load balancing in CDN [131].
Figure 5.1 shows the layered architecture of CDN system with application layer at the top at client end, then the type of CDN services offered. Third layer is management layer which is responsible for resource allocation security and load balancing and all other lower layers are standard network layers protocols and connectivity layers. Servers are the last entity which remains connected.

Fluid models are being proposed for TCP flow control in many MANET routing protocols [172]. Proposed CDN framework consists of servers with independent queues of self-determining queue length and service rate. Our proposal uses a fluid model for dynamic queue and real time behavior of the system. We have assumed a CDN with ‘n’ number of servers having service queue and high rate of request traffic over the system, which cannot be fulfilled by single system and the system remains in critical condition [173]. Load balancing plays an important role to resolve the critical condition of servers by diverting requests to servers with lower request rate and empty queue which can serve the requests. For a server with a fluid flow model Figure 5.2, we need to introduce few notations.
\( Q_i(t) \): Queue length of server ‘i’ at time t.

\( \alpha_i(t) \): Request arrival rate of server ‘i’

\( \mu_i(t) \): Service rate of the server ‘i’

Fluid model can be defined by equation (5.1)

\[
\frac{dQ_i(t)}{dt} = Q_i'(t) = \alpha_i(t) - \mu_i(t) \tag{5.1}
\]

where, \( i = 1, 2, \ldots \)

\( Q_i'(t) \) are the extra requests to be serviced. In a fluid model, if service rate is less than the request arrival rate \( \alpha_i(t) \), the queue size increases. Consequently requests in the queue have to wait for a longer time. Additionally, with increase in queue length, load over the server increases resulting in an increase in response time and computation time. As a result, the probability of request failure also increases. To provide best QoS (Quality of service) we need to maintain the relation between the average incoming rate and average service rate as given in equation (5.2).

\[
\bar{\alpha} \leq \bar{\mu} \tag{5.2}
\]

where, \( \bar{\alpha} = \) Average incoming rate

\( \bar{\mu} = \) Average service rate

Hence, we need to find out the probability of request failures in server over a time period t. It is assumed that request rate \( \alpha_i(t) \) follows Poisson’s distribution as per equation (5.3).

\[
p(x, \lambda) = \begin{cases} 
\frac{\lambda^xe^{-\lambda}}{x!}, & x \geq 0 \\
0, & x < 0
\end{cases} \tag{5.3}
\]

where,

‘e’ is the natural logarithm.
‘x’ is the number of events in a given interval,
‘λ’ is the mean number of events per interval (a positive number representing the expected number of occurrences within a specified interval).

For example, if 6 requests arrive every 10 minutes, then for 1 hour λ will be 36. The Poisson distribution models the occurrence of events without knowing the total number of possible occurrences.

Poisson distribution is used for calculating fault rates and reliability of a system.

So probability distribution for failure in a system can be given by equation (5.4)

\[
f(x, \lambda) = \begin{cases} 
\frac{\lambda^x e^{-\lambda}}{x!}, & x \geq 0 \\
0, & x < 0
\end{cases} \quad \ldots (5.4)
\]

Equation 8 shows the failure probability distribution over time t, x (number of failures), λ (failure rate). Where F (T) is defined as the probability of failure over the time t. To define failure in a system over a time t and t+ Δ T is given by equation (5.5):

\[
f(t \leq T \leq t + \Delta T | T > t) = \frac{\exp(-\lambda t) - \exp(\lambda (t + \Delta T))}{\exp(-\lambda t)}
\]

\[
= 1 - \exp(-\lambda t) \quad \ldots (5.5)
\]

\[
f(t) = 1 - \exp(-\lambda t), \quad \text{for interval [0, t]}
\]

Reliability of a system can be defined in terms of many parameters such as durability, failure and QoS over a time t. In general, reliability can be defined as the probability of an item to perform a required function under stated conditions for a specified period of time.

The Probability that a system is reliable over a time t can be given by equation (5.6):

\[
R(t) = 1 - f(t) \quad \ldots (5.6)
\]

\[
= 1 - \exp(-\lambda t) \quad \ldots (5.7)
\]

\[
= \exp(-\lambda t) \quad \text{for interval [0, t]}
\]

And for time interval [t, t+ Δ T] reliability R (Δt) is given by equation (5.8)

\[
R(t) = \exp(-\lambda t)R(\Delta t)= \exp(-\lambda t) - \exp(-\lambda (t + \Delta T)) \quad \ldots (5.8)
\]
5.2 PROPOSED ALGORITHM

Proposed load balancing algorithm is an improvement over existing algorithms discussed above [51][104][134][136]. It takes into account the faults occurring in a server over a period of time. Proposed algorithm is based on the following parameters:

**System Load:** Percentage of request queued up in server queue.

**Fault rate:** Faults occurring at the datacenters per unit time.

**Queue Size:** Maximum requests that can be accommodated in the server queue.

**Response time:** Time required to start handling a request.

**Network load:** Total bandwidth utilized by the server.

Each server is assumed to have its own service rate, request rate, response time, queue size and failure rate. This may keep on varying dynamically with time. For balancing the load, overloaded servers are required to be identified. A Server is said to be overloaded, if:

\[ Q_{i\text{max}}(i) < Q_{i}(t) \]  \hspace{1cm} \text{... (5.9)}

where,

- \( Q_{i\text{max}}(i) \): Maximum queue size of server i;
- \( Q_{i}(t) \): Queue length of server i at time ‘t’
- \( \Delta Q_{i} \): Extra Queue to be balanced

Proposed algorithm is divided into three phases:

a) Initialization

b) Load balancing

c) Updating
5.2.1 Initialization

All parameters are initialized and investigated with default values and updated periodically. Fault rate and Network load is initialized to zero. Queue size and response time are based on individual server’s specifications. Fitness is calculated based on these values. Newly introduced servers are always initialized with default values. Further, the fitness value is iteratively calculated and updated periodically.

Initial parameters are defined as:

- **Fault_Ini**: Initial fault rate.
- **Q_Size (i)**: Initial Queue length on server ‘i’.
- **Resp_Ini**: Initial response time on server ‘i’.
- **N_load_Ini**: Initial network load.
- **S_load_Ini**: Initial system load.

5.2.2 Load balancing

A server having full queue cannot accommodate additional requests and some requests in queue may fail due to deadline constraints. In such circumstances load balancing is essential to avoid/reduce request failures.

To overcome this replica of the data being requested is made on another server to balance the request load over the original server. To balance load a server which can fulfill the request with highest fitness value and same quality of service as promised by original server has to be identified. Here we can classify the servers into two categories as hot spot and cold spot.

Hot spots are those servers which are overloaded with requests and have most of the MIPS and network bandwidth utilized and have long request waiting queue. Cold spots are those servers which have low request rate and can accommodate more requests. Load balancing is required to stop servers from becoming hotspot and find a cold spot to balance the request load. Whenever a server is found over loaded based on equation 11 we need to find server which can fulfill extra requests defined by equation (5.10):
\[ \Delta Q_i = Q_i(t) - Q_{\text{Size}(i)} \quad \ldots \quad (5.10) \]

\( \Delta Q_i \) is the extra queue size to be balanced on server \( i \) where \( i \in \{1, 2, 3 \ldots n \} \). If \( \Delta Q_i \) is positive we will call load balancing function. To balance the load we need to find a server with empty queue length and highest fitness value from a list of all servers maintained. This list is used by load balancing algorithm along with other parameters to find the best server over which request can be diverted.

- **Fault rate**

Fault rate is directly proportional to the server load, network load, system load. This implies that more the load on the server, the more is the probability of failure. This is because the queue size becomes very large. If the size of the queue is beyond the processing rate, the requests waiting time increases which lead to request failure. On the other hand system load also increases the probability of system failure in the form of hard disk and machine failure.

\[ \lambda(t) = f(N_{\text{Load}}, S_{\text{load}}) \quad \ldots \quad (5.11) \]

\( \lambda(t) \): fault over a time \( t \).

Equation (5.11) says that fault rate is directly proportional to system and network load.

\( \lambda \): fault rate

\[ \lambda = \Sigma \text{total number of fault} / \text{per hour}, \quad \ldots \quad (5.12) \]

- **Response time**

It is the time duration between request submission and starting of processing. It is directly proportional to the total load on the system. Server with least average response time should be chosen.

The Fitness value for a server can be determined as:

Fval \((s)\): Fitness value of server \( s \) can be given by equation (5.13)

\[ \text{fval}(s) = \left( \alpha_1 * \frac{1}{\lambda} \right) + \left( \alpha_2 * \frac{1}{\text{Resp}} \right) + \left( \alpha_3 * \frac{1}{N_{\text{load}}} \right) + \left( \alpha_4 * \frac{1}{S_{\text{load}}} \right) \]

free(Queue\_Length)

\( \ldots \quad (5.13) \)

where,

\[ \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 = 1 \]
Load balancing is divided into 3 steps:

- **Step 1:** Find the list of all the servers which have empty queue greater than $\Delta Q_i$.

- **Step 2:** From this list find a server for load balancing with highest fitness value but at the same time the new server should have less or equal fault rate than the searching server to provide same or higher quality of service as promised by the original server, i.e. least fault rate, lease network load, least system load, and largest free queue length.

- **Step 3:** Transfer the set of extra requests to the selected server.

### 5.2.3 Updating Algorithm

It is an iterative process after uniform time interval for updating the current status. This phase is repeated after an equal interval of time to get the updated server status. Initially default value zero is assigned for network load, system load and fault rate $\lambda(t)$. The Queue length of a server is always initially zero because there is no request made to that server.

\[
\lambda(t)_\text{Initial} = 0 \quad // \text{Initial fault rate}
\]

\[
N\_\text{load}_\text{ini} = 0 \quad // \text{Initial network load.}
\]

\[
S\_\text{load}_\text{ini} = 0 \quad // \text{Initial system load.}
\]

\[
Q\_\text{len}_\text{ini} = 0 \quad // \text{Initial queue length}
\]

\[
\text{Res}_\text{Ini} = \text{Non zero} \quad // \text{Initial server response time}
\]

New fitness values are calculated based on the changes occurring in existing values. Suppose that ‘Si’ is the server,

\[
\lambda(t)_\text{new} = \text{New Fault rate over time ‘t’}
\]

\[
N\_\text{load}_\text{new} = \text{New network load,}
\]

\[
S\_\text{load}_\text{new} = \text{New system load,}
\]

\[
Q\_\text{len}_\text{new} = \text{New queue length}
\]
fval_new = New fitness value of server i, ‘Si’

Then New Fault rate over time ‘t’ is given by:

\[ \lambda (t) = f\left(N_{Load_{new}}, S_{load_{new}}\right) \]  
... (5.15)

\[ Fval_{new}(s) = \left(\alpha_1 \ast \frac{1}{\lambda_{new}}\right) + \left(\alpha_2 \ast \frac{1}{Resp_{new}}\right) + \left(\alpha_3 \ast \frac{1}{N_{Load_{new}}}\right) + \left(\alpha_4 \ast \frac{1}{S_{Load_{new}}}\right) + \text{free}(Queue\_Length) \]  
... (5.16)

where, server_id = min (fval1new, fval2new,..., fvalnnew)  
... (5.17)

\[ \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 = 1 \]  
... (5.18)

\[ \alpha_1 > \alpha_2 > \alpha_3 > \alpha_4 \]  
... (5.19)

After updating the fitness value, next incoming request is allocated to the server having highest fitness value according to equation (5.16).

---

**Load balancing Algorithm**

1: Initialize servers
2: Start sending requests
3: push request in queue.
4: if (queue length > server queue length)
5: \[ s = \text{find\_server()} \]  // find server with empty queue and highest fitness value
6: if (( s ! = searching\_server ) & (fault rate < searching\_server))
7: Migrate request = > “s”
8: else
9: keep searching free server.
10: else
11: pop request from queue process it

---

**Update fitness value algorithm**

1: Find updated values of parameters
2: Find network load
3: Find system load
4: Find fitness value using equation 14
5: update the new fitness value
5.3 EXPERIMENTAL RESULTS

Simulation is carried out on GridSim [174]. By default, GridSim does not have any feature to recognize failures in servers. Hence a module is developed and added to GridSim that supports fault aware scheduling. This helps in study of the performance of CDN in the fault aware environment. To create network architecture of Figure 5.1, Brite file is used as simulation data. Brite file helps to define network properties and interconnection between the nodes which are servers in our case. To compare performance based on the number of faults occurring by using each of the previous algorithms and proposed algorithm, we have considered 3 servers S1, S2, S3 each having independent failure rate \( \lambda (t) \), request arrival rate, processing rate and queue length. Table 5.1 shows the specification of each server.

<table>
<thead>
<tr>
<th>Server Name</th>
<th>Fault rate</th>
<th>Service rate</th>
<th>Queue length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server1 (S1)</td>
<td>0.143</td>
<td>7</td>
<td>20</td>
</tr>
<tr>
<td>Server2 (S2)</td>
<td>0.125</td>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td>Server3 (S3)</td>
<td>0.5</td>
<td>6</td>
<td>50</td>
</tr>
</tbody>
</table>

Queue length defines that after the specific queue is full extra requests will be balanced using proposed algorithm, to avoid request failure due to large waiting time. Table 5.2 and Figure 5.3 show the number of requests failed when the algorithms are tested for 60,100, 200,500,600 and 700 requests with all algorithms. Workload traces are achieved from load traces of DAS-2 multi-cluster system obtained from the Parallel Workload Archive are used to generate requests [175].
Table 5.2: Request Failure Count

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Number of request counts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60</td>
</tr>
<tr>
<td>Proposed</td>
<td>8</td>
</tr>
<tr>
<td>QLBLB</td>
<td>13</td>
</tr>
<tr>
<td>RAND</td>
<td>12</td>
</tr>
<tr>
<td>2RC</td>
<td>10</td>
</tr>
<tr>
<td>LL</td>
<td>10</td>
</tr>
<tr>
<td>RR</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 5.3: Failure count of proposed algorithm against other algorithms

To compare performance based on probability of failure occurring by using each of the previous algorithms and proposed algorithm using scenario given in Table 5.1.

Table 5.3: Probability of request failures

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of request counts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60</td>
</tr>
<tr>
<td>Proposed</td>
<td>0.133</td>
</tr>
<tr>
<td>QLBLB</td>
<td>0.216</td>
</tr>
<tr>
<td>RAND</td>
<td>0.75</td>
</tr>
<tr>
<td>2RC</td>
<td>0.166</td>
</tr>
<tr>
<td>LL</td>
<td>0.166</td>
</tr>
<tr>
<td>RR</td>
<td>0.166</td>
</tr>
</tbody>
</table>

Table 5.3 and Figure 5.4 show the probability of request failure when each of the algorithm is tested over 60, 100, 200, 300, 400, 500, 600 and 700 requests. Table 5.3 shows that with increase in request failure probability increases for QLBLB, 2RC and LL. On the other hand the probability of failure is stable for RR and Rand but greater then proposed algorithm. This shows that the proposed algorithm proves to have a lower failure count and failure probability as compared to other algorithms. Proposed algorithm demonstrated a lower failure count.
percentage in between 13% to 17% when number of requests were increased from 60 to 700, as compared to other existing algorithms whose failure counts percentage were up to 23% for QLBLBA. Other existing algorithms like RAND, 2RC, RR and LL algorithms also had higher failure counts than the proposed algorithm as shown in Table 5.3.

![Figure 5.4: Failure probability of proposed algorithm against other algorithms](image)

The Proposed algorithm can also be compared with other algorithms based on one more parameter, i.e. reliability which is defined in equation 9. Reliability defines the algorithm to be more dependent and probability that the request will be completed. So, higher the reliability lower the chances of request failure.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>60</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>700</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed</td>
<td>0.867</td>
<td>0.85</td>
<td>0.86</td>
<td>0.84</td>
<td>0.83</td>
<td>0.83</td>
<td>0.83</td>
<td>0.8271</td>
</tr>
<tr>
<td>QLBLB</td>
<td>0.784</td>
<td>0.77</td>
<td>0.77</td>
<td>0.767</td>
<td>0.776</td>
<td>0.78</td>
<td>0.7816</td>
<td>0.7614</td>
</tr>
<tr>
<td>RAND</td>
<td>0.25</td>
<td>0.75</td>
<td>0.82</td>
<td>0.82</td>
<td>0.818</td>
<td>0.82</td>
<td>0.8183</td>
<td>0.8114</td>
</tr>
<tr>
<td>2RC</td>
<td>0.834</td>
<td>0.81</td>
<td>0.82</td>
<td>0.801</td>
<td>0.803</td>
<td>0.8</td>
<td>0.805</td>
<td>0.7971</td>
</tr>
<tr>
<td>LL</td>
<td>0.834</td>
<td>0.77</td>
<td>0.805</td>
<td>0.8</td>
<td>0.796</td>
<td>0.7933</td>
<td>0.7842</td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>0.834</td>
<td>0.82</td>
<td>0.815</td>
<td>0.8134</td>
<td>0.815</td>
<td>0.814</td>
<td>0.8133</td>
<td>0.8128</td>
</tr>
</tbody>
</table>
Figure 5.5: Reliability of proposed algorithm and existing algorithms

Table 5.5: Completed request count

<table>
<thead>
<tr>
<th>Alloys</th>
<th>Number of request counts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60</td>
</tr>
<tr>
<td>Proposed</td>
<td>52</td>
</tr>
<tr>
<td>QLBLB</td>
<td>47</td>
</tr>
<tr>
<td>RAND</td>
<td>48</td>
</tr>
<tr>
<td>2RC</td>
<td>50</td>
</tr>
<tr>
<td>LL</td>
<td>50</td>
</tr>
<tr>
<td>RR</td>
<td>50</td>
</tr>
</tbody>
</table>

Figure 5.6: Completed request count of proposed algorithm with existing algorithms
Table 5.4 and Figure 5.5 show the increase in reliability using proposed algorithm and improvement over other algorithm. Other advantages of the proposed algorithm that can be derived from Table 5.3 and Table 5.4 is that the algorithm which has higher reliability has shown to have high request failure, on the other hand proposed algorithm have a lower request failure and higher reliability. Table 5.5 and Figure 5.6 show the improvement in Count of completed requests using proposed algorithm over existing algorithms in faulty environment.

\[
\text{Average Queue length} = \frac{\sum_{i=0}^{n} \text{Max\_length\_i}}{n} \quad \text{...(5.20)}
\]

where, \( n \) = number of servers,

\( \text{Max\_length\_i} = \text{Maximum queue length of server } i \)

Proposed algorithm is also compared on the basis of maximum queue length, given by equation (5.20). Higher the queue size, more is the waiting time for requests. This increases the probability of request failure over period of time. Comparing the maximum queue length achieved by each algorithm, the one with minimum queue length can be considered as best algorithm.

![Figure 5.7: Queue length of 2RC and QLBLB algorithm](image)

(a) 2RC  (b) QLBLB

![Figure 5.8: Queue length of Proposed Fault and LL algorithm](image)

(a) Fault aware  (b) LL

**Figure 5.7: Queue length of 2RC and QLBLB algorithm**

**Figure 5.8: Queue length of Proposed Fault and LL algorithm**
Figure 5.9: Queue length of RAND and RR algorithm

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>RR</th>
<th>LL</th>
<th>RAND</th>
<th>2RC</th>
<th>QLBLB</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Queue length</td>
<td>96</td>
<td>101</td>
<td>101</td>
<td>150</td>
<td>129</td>
<td>87</td>
</tr>
<tr>
<td>Failure count</td>
<td>97</td>
<td>95</td>
<td>96</td>
<td>94</td>
<td>97</td>
<td>93</td>
</tr>
</tbody>
</table>

Figure 5.7, 5.8 and 5.9 shows the behavior for queue length due to proposed and all other algorithms. An observation that comes out from above figures is that the algorithm which has lower average queue length but has a higher fault rate. Figure 5.7 (a) of 2RC algorithm has 150 average queue length and so on for other algorithm as shown in table 5.6 correspondingly. Table 5.6 clearly shows that an algorithm which has a lower average queue length, but has a higher failure count like RR algorithm, but proposed algorithm prove to have better performance in term of average queue length and failure count at the same time compared to other algorithm. Output for Table 5.6 is tested in the scenario shown in Table 5.1 with 3 servers and corresponding failure rate, service rate and queue length. The service rate for server 1 is 7 requests, i.e., it can process 7 requests at a time, similarly 6 requests for server 2 and server 3.
Table 5.7: Throughput

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Request count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60</td>
</tr>
<tr>
<td>Proposed</td>
<td>86.6666</td>
</tr>
<tr>
<td>QL_BLB</td>
<td>78.3333</td>
</tr>
<tr>
<td>RAND</td>
<td>80</td>
</tr>
<tr>
<td>2RC</td>
<td>83.3333</td>
</tr>
<tr>
<td>LL</td>
<td>83.3333</td>
</tr>
<tr>
<td>RR</td>
<td>83.3333</td>
</tr>
</tbody>
</table>

Figure 5.10: Throughput Comparison of proposed algorithm against other algorithms

Table 5.7 compares the throughput of proposed algorithm and other algorithms for 60, 100, 200, 300, 500, 700 requests over the servers. Figure 5.10 compares the throughput of proposed algorithm graphically and shows the improved throughput of proposed algorithm over other algorithms. Taking into consideration all the performance parameters we can suggest that fault and reliability based proposed algorithm prove to have better performance and QoS over other algorithms.
5.4 CONCLUSION

In this chapter, different types of Load balancing algorithm have been discussed with their drawbacks in CDN. To overcome the drawbacks, an efficient fault aware load balancing algorithm is proposed which performs better than other existing load balancing algorithms for CDN in the fault aware environment. For future work, this algorithm may be compared with other proposals and study may be done for further improvements in the QoS.