J. Appl. Environ. Biol. Sci., 7(12)201-214, 2017

ISSN: 2090-4274
Journal of Applied Environmental and Biological Sciences
www.textroad.com

Implementation of Multiple Thread Pools Based on Distribution of Service Times

Riaz Shah*, Faisal Bahadur, Noor-ul-Amin, Arif Iqbal Umer, Muhammad Javed Shad

Department of Information Technology, Hazara University, Mansehra, Pakistan

Received: July 27, 2017
Accepted: September 25, 2017

ABSTRACT

The demand of internet is increasing day by day and web server receives millions of hits per day. To manage these raising demands high performance web servers are required. Multithreading is the elementary approach used in web server to achieve high performance and handle number of requests from clients. The existing scheme which is distributed frequency based optimization strategy DFBOS cannot distribute thread pools on the basis of service times due to this starvation occurs. When starvation occurs it decreases response time and increases wait time. The research aims is to explore the implementation of multiple thread pools based on distribution of service times to avoid starvation and achieve concurrency in server site. For comparing both strategies we have used a simulator named as Thread Pool Tester TPT which is a JAVA based simulator and it has shown that proposed strategy is much better than the existing DFBOS. The analysis shows that proposed scheme is increases the response time and reduces the wait time.

KEYWORDS: Thread Pool Allocator, Slave Listener, Request Listener, Concurrency, Thread Pool, Multithreading, Threads.

1.INTRODUCTION

To achieve concurrency at server side application we use multi threading. In similar programs various activity can be processed in multithreading. Multi threading execute multiple process in multiple CPUs, therefore provide enhance application retort.

Mostly used techniques to increase throughput of CPUs is multithreading. Thread-pool administration is considered one of the challenging job in this method since it require sufficient numbers of thread in the thread-pool tuning system which can provide better responsive time and soaring utilizations of scheme possessions [14].

Multithreading construction used systems resource competently and as well as provides multi processor architecture hence it has become the supreme option for execution of server side applications. Although require run times process of memory allotment and de allotment for threads creation and distortion [10].

Thread pool and thread per request is the most commonly used multithreading architecture. The both architecture has its own advantage and disadvantage. Now we are explaining it briefly as following.

In thread per request architectures for every client request a thread is created. After the completion of request the thread is destroyed. The advantages of these architectures are that it is easily to implement as compare to other architecture. This strategy can be used for task which has extensive running time like database query which is coming from a number of users on different servers. The drawbacks of this strategy are that it consume system properties if continue numbers of customer are convey demand to server.

In thread-pool architectures thread are generated in the initial stage of thread-pool-system. That structural design is beneficial for ORBs in which numbers of resource can be utilized by OS and kept initially. In this strategy request is processed continuously although if the numbers of request on server surpass then the existing thread in the pool then the exceeding task has to stay for execution in the queue. The request may be either in ready or in waiting condition. The benefit of this strategy in multi-threading architectures is that it is simple to execute. The drawbacks of this strategy are unnecessary framework changing and synchronization complexity. Although threads pool as compare to thread-per-request architecture enhance system throughput and decrease reaction times for users [2].

1.1. Consequence of Implement Multiple Thread Pools Based on distribution of Service Times for DFBOS

The existing strategy DFBOS equally is distributed the request among the different node's Increase the node's performance through the load balance mechanism. In DFBOS they cannot distribute thread pools on the basis of service time. In DFBOS starvation can occur which decrease the performance of system.

Corresponding Author: Riaz Shah, Department of Information Technology, Hazara University, Mansehra, Pakistan

In Our proposed strategy we can implement Multiple Thread Pools Based On distribution of Service Times for DFBOS, Which can avoid the starvation and also improve the performance of the system. Research question for the proposed scheme is as under.

i. How to remove starvation of short tasks in a Multiple Thread pool environment?

1.2. Assessing Effect of Implementation of Multiple Thread-Pools Based On distribution of Service-Times for DFBOS

For using the effect of implementation of multiple thread pools based on distribution of service times for DFBOS, we use the following methodology.

i. Reuse the current link queue of JAVA 5 for Implementation of Multiple Thread Pools Based On distribution of Service Times for DFBOS.
ii. Request counter can be used for CAS code arrangement for shared resources.
iii. Code can be constructed for Implementation of Multiple Thread Pools Based On distribution of Service Times.
iv. For performance association we use the simulator named as thread pool tester which is based on java applications for our proposed strategy that is the implementation of Multiple Thread Pools Based On distribution of Service Times with existing strategy DFBOS.

2. RELATED WORK

In [1] developed a thread pool system which consists of stable number of threads in the pool. In this thread pool at the start the thread per request replica are used. They consist of queue in which the incoming requests from clients are inserted in first in first out order (FIFO). The threads which are exist in the thread pool pick up these request and when the number of request are increased then the available threads in the pool, then more threads are generated in the thread pool until it reached to the soaring water mark , and then no more threads can be created.

In [3] provide COBRA stipulated multithreaded ORB foundation thread pool system. COBRA is an object oriented atmosphere which can propagate over the computer network for computing. The numbers of threads in the thread pool are fixed and the threads are used to process the incoming client's requests. The request can be received from different users over the network and it can process continuously. The extra request from clients can be stored in queue and it can be executed when the threads are variables.

In [5] they both present a thread pool for COBRA real time requirements. On the other hand when the numbers of request are increased at higher level than the thread pool cannot grow further called the high watermark, the thread pool in which the number of thread cannot grow further is called bounded thread pool. In real time COBRA thread pool the threads giving the prioritization that is the threads can be divided into partitions.

In [6] present the thread pool for internet of things (IOT), they can socket the server to handle the simultaneous clients request on server. The communication layers generate the large numbers of threads to process the simultaneous incoming request from users. This thread pool model creates two pools to handle the large number of request one pool for the incoming clients request the other for the processing of request.

In [7] presented a thread pool model for energetically determine the best possible thread pool size and they can used the heuristic techniques. They can examine the interior attributes of thread pool system and they can also observe the performance of thread pool model by using various multithreaded structural design. In this thread pool model they can calculate the average idle time (AIT) of the requests. The average ideal time is calculated after the completion of five tasks, and the increase the thread pool size after the execution of five tasks then the average ideal time is increased by one percent. Without any justification they can used the different values in algorithm.

In [8] proposed a thread pool model in which the threads are borrowed for other pools, and they can use the heuristic computations. The measured the average ideal time for those jobs which have in waiting state after about 20ms, then he compare the average ideal time for all threads which are in waiting queues and then from the less thread pool system they can get borrow. Here they can measure the average ideal time of the waiting threads but they show the lower performance.

In [9] proposed a thread pool system for online server applications. For proficient resource consumption they can use some extra factors in the thread pool system to adjust the pool size. They can calculate the average wait time of the queue request and increases the thread pool size at the extend when the wait time become turn down. The drawbacks of this paper is lower response time recorded by clients and when they can measure the average ideal time then the ready queue become padlock.

In [10] proposed the most favorable size for thread pool model. For efficient threads organization they can used the numerical examination and the can calculate the most favorable pool size by using the arithmetical and numerical calculations. To measure the favorable pool size they can use the thread background control time and creation time of thread. The drawbacks of this paper are it is difficult to measure the current parameters at kernel level and still it can't be measured by any web server.
In [11] presented a method which can be used for analyzing and assigning the resource management called model fuzzing. They can also calculating the output criteria at different stages, and also analyze the concurrency level for different numbers of threads. They can use the model for analyzing the concurrency level by using different parameters. They can record the different alternatives of the current working system and the remaining alternatives and then compare with each other and provide the performance of the system. They can predict that the workload of the incoming and outgoing requests demonstrate diverse kind of attribute at diverse times, which is concluded a wrong prediction.
In [12] proposed gaussian division for manipulating prospect guess. the request are coming the thread pool creating the threads automatically and when the numbers of request are decreased then it delete the threads automatically means that the thread pool system work dynamically. The prediction is unsuccessful at high and low request rate.
In [14] presented a guessing system named as narrative tendency exponential moving average system (TEMA) to elaborating the EMA. They want to avoid the duplication of threads from EMA, by using arithmetical hypothesis they can calculate the rate of change of threads in the thread pool by prediction.
In [15] proposed a thread pool model for cluster atmosphere. They can use for calculating the concentrated requests. In this model we have to add more machines in the cluster, there is no need of cluster alteration. For the efficiency of thread pool they can use the load balancing algorithm. The drawbacks of this system are to load balance between the different nodes, and also added the new nodes.
In [16] proposed a thread pool model for disperse cluster atmosphere. The give the idea of perpetrator service which is disperse and which process a submitted task and control it by using mobile controller. The thread pool system consists of threads which is independent of each other and executed within pool. Here they can use load balancing by using mobile controller. The drawbacks of the paper are there in no archetypal confirmation.
In [17] proposed a thread pool model called FBOS. Which compute the incoming client's requests, and it can decrease or increase the thread pool system automatically when the requests are coming. They can calculate different parameters such as response time, wait time and turnaround time of the request. The drawback of this paper is that they can use the locks which can slow the performance of the system.
In [18] proposed a thread pool system which can distribute the load among different node by using the load balancer, the drawbacks of this paper is that they cannot distribute the thread pool on the basis of service time due to which starvation occurred.

2.1. The drawbacks of the existing system

i. The can only distribute the work load among the different.
ii. They cannot distribute the thread pool on the basis of service times.
iii. Starvation occurs which can decrees the performance of the system.
iv. Context switch and synchronization overheads.

To overcome the drawbacks of the existing DFBOS we implement multiple thread pools based on distribution of service times to avoid starvation and achieve concurrency in server site.

3. MATERIAL AND METHODS

In this section we will discuss about the proposed system named as implementation of multiple thread-pools based on distribution of service-times.

3.1.Assumption for Thread Pool System

i. Threads have the same precedence in the thread pool. In thread pool system each thread consumes the same amount of CPU time. In thread pool system there is no classification of clients i.e. client or supervisor, every clients have the same priority in the thread pool system.
ii. In our proposed strategy we can implement Multiple Thread-Pools Based on distribution of Service-Times for DFBOS. This can avoid the starvation and improve the performance of the system.

3.2. Existing DFBOS Scheme

In figure 1 shows the existing scheme i.e. DFBOS (Distributed Frequency Based Optimization strategy) it consists of job in queue which stockpiles requests coming from user side and hold by a dynamic job in queue. This strategy

is used for first in first out (FIFO) data structures in which the requests are entered from one site and taken from other site of the queue after the execution. After the job in queue the request is hold by load balancer thread pool consists of threads which are used to fulfill the client's requests. Load balancer thread pool is connected to distributed slave server thread pool. When the requests are coming from users the load balancer thread pool divides the load equally to itself and also to the distributed slave serve thread pool. It can distribute the same workload among the slave nodes. After the completion of task it can be store in job out queue. Through the load balancer mechanism every node shows its performance. Load balancer divides the workload among the different nodes through the help of global table, listener and register.

Listener is the pathway from loader balancer to the other slave node, and also it detects the new incoming task which is coming from user. It also sends demand to load-balancer for the addition or removing of new node. If new node is added to the load-balancer then it added them to the register and calls to global table.

Register are used for the registration of new node when the listener listen about the addition of new to the load balancer. Then it call to the global-table for the verification of new node added or not, if the new node is added to the load balancer then the global table generates a specific IP for the already added node.

Global table consists of IP's of all nodes that are connected to the loader balancer, if the nodes are added or removed from load balancer it can maintain the IP's of that nodes. It can update the IP's of nodes.

The drawbacks of the existing method are that they can only distribute the work load between the various nodes without considering service times, in the results of that it occurs starvation .i.e. if we have different nature of jobs such that light and heavy weight if the heavy weight job acquire the CPU then the light weight job may wait until the processing are finished. To avoid starvation we can introduce a new technique in the next section i.e. implementation of multiple thread-pools based on distribution of service-times.

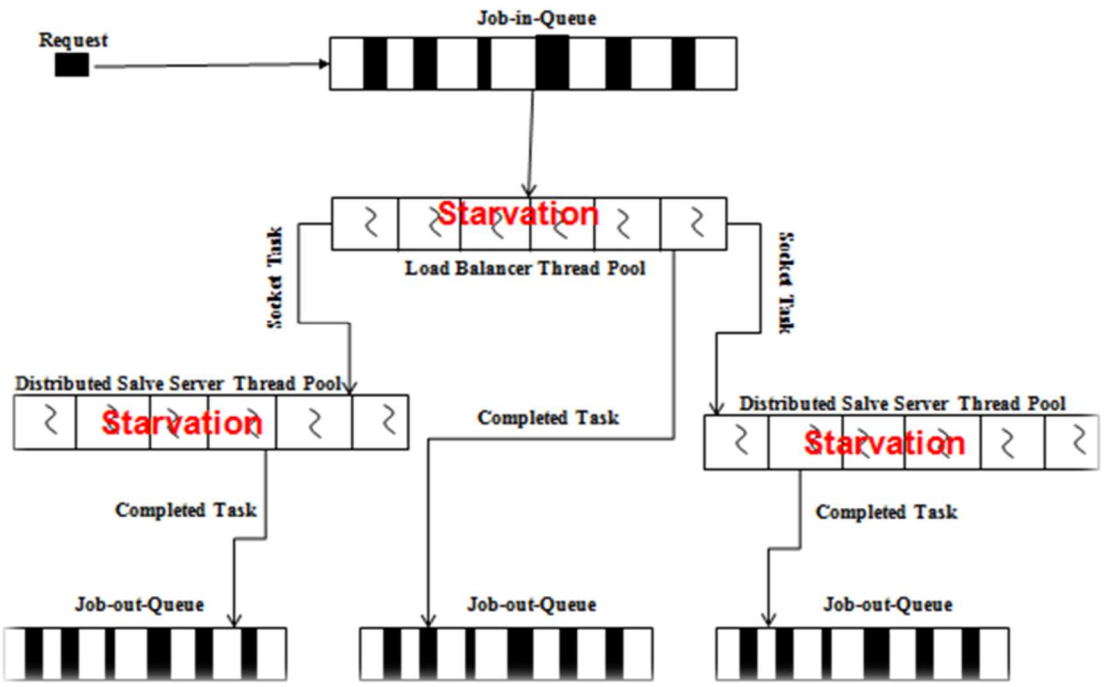

Figure.1: Task Distribution without considering service time in DFBOS

3.3. Implementation of Multiple Thread-Pools Based on Distribution of Service-Times

Before the start of proposed scheme, first we discuss the limitation of existing strategy that they can distribute the work load through load balancer among the different nodes without considering service times in the result of which the starvation occurs. Figure 2 shows the implementation of multiple thread-pools based on distribution of service-times for DFBOS in which we can distribute the thread pool based on distribution of service times. In this scheme we can use the different nature jobs that is light and heavy weight jobs e.g. if we have a jobs which processing times is 100ms and the other one which have 200ms if the 200ms job acquires the CPU then the 100ms job will wait for

200ms for other jobs and 100ms for itself processing time then here the starvation will occur. To avoid starvation we can divide the thread pool in to low service time thread pool (100ms) and high service time thread pool (200ms). For validating our strategy we can use a simulator named as thread pool tester (TPT). TPT consist of two main agents that are client tire and the server tire. In server tire we can install the simulator and then embed our proposed thread pool. Before the addition of any clients we can run the main server switch for offline workload profiling table. And store they require workload in workload profiling table. The status of jobs is non active initially. After WP table we can initiate the slave listener for checking the new node if they can detect the new node then they can update the thread pool table. Initially we have only one thread pool in TP table i.e. local pool, the slave listener work continuously for detection of new node. After the TP table we can initiate the request listener and it work continuously for detection of new request from nodes, if they can detect the new jobs then they can check it in workload profiling table for their service times and then send it to the required thread pool. In proposed scheme we can used two nature of jobs i.e. 100ms and 200ms, and also we can used the two pools one for 100ms and the other for 200ms.The proposed scheme improves the performance of system.

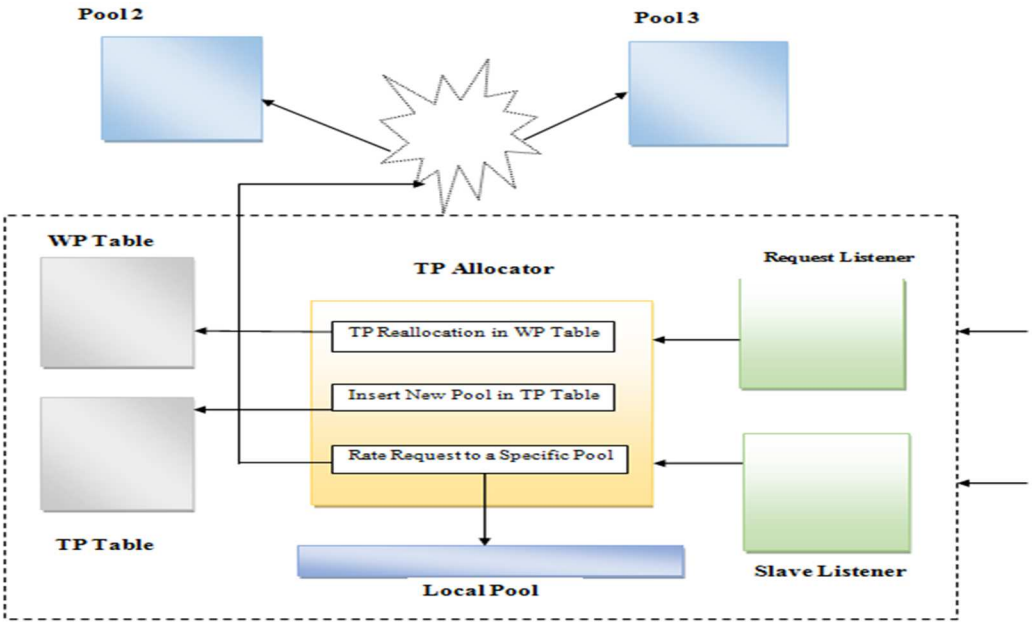

Figure.2: Block Diagram for Proposed Scheme

The proposed strategy is consisting of numerous classes. Here we can introduce the factors that are used in our proposed strategy one by one.

Offline Workload Profiling Table: WP table is used to store the service time, status and the thread pool of jobs. Before the addition of any node we can run our main switch for storing the service times of jobs in WP table. Initially, the status of every job is non active and they can reside in local pool. Initially we have only two types of jobs that are 100ms and 200ms. When they request listener detect the jobs from the node, then they can check it in WP table for their service time and then sent it to the required pool. After the detection of jobs its status will become active and also its thread pool will be changed in WP table. The jobs are in sorted order in WP table, Workload Profiling table as shown bellow:

Jobs	Service Times	Status	Thread Pool
Job1	100ms	Non active	1
Job2	200ms	Non active	2

Table.1: Offline Work Load Profiling Table

Thread Pool Table: TP table consists of pools. Initially it consists of one local thread pool and when the new server are add to the system it can also added a separate pool for it. We can also check that thread pool is stable or not. TP table is demonstrated as follows.

Pool	IP	Stability
1	Local host	Yes
2	192.48.45	Yes

Table.2: Thread Pool Table

Slave listener: Slave listener listens the port thoroughly for new node if the new node is detected then it update the thread pool table and then passes it to the thread pool allocator. It can add new thread pool in the TP table.
Request Listener: Request listener listens the incoming jobs from the clients thoroughly. When the jobs are detected it can update the work load profiling table and pass it to the thread pool allocator. Then the TP allocator allocates the thread pool for the new incoming jobs on the basis of service times.
Thread Pool Allocator: The main agent of our proposed scheme is the TP allocator. It can call from two sides that is slave listener and from request listener. The TP allocator performs three main functions. It can reallocate the thread pool in work load profiling table. Second it can add a new thread pool in thread pool table when the new slave nodes are detected. The last one it can send a job to a specific thread pool when the incoming requests are detected.
Local Thread Pool System: The object of this class is initiated first. Our overall thread pool system is represented by this class. The object of this class initializes a whole system which is important for our thread pool system. When scheme starts it can create an instant of job-in-queue which can store the incoming user's request. When the user requests are coming for the first time it can process it in the default thread pool and also store its value in log table for further use, when the requests are for the first time from user it can also check it in log table for finding its processing time. If the same job is processed, before it can then send it to the required thread pool for further processing.

a. Main Server Algorithm Flowchart

When the main switch server are started for validating the proposed scheme it can initiate the work load profiling table for storing the service time, status and pool of the jobs. Then it can initiate the thread pool table. When the new node is added to the main server, it can add the new thread pool for it in TP table. The TP table updates continuously, it inserts new slave server. It starts the thread pool table which has only one local thread pool, which relates to the main server switch. After TP table it can launch slave listener for detecting the new slave node thoroughly. If they detect the new slave server then it can add a new thread pool for it in TP table. At second last it can launch the request listener for detecting the new request from clients thoroughly and can update the. At last the main server can initiate the local pool. WP table and then passes it to the TP allocator.

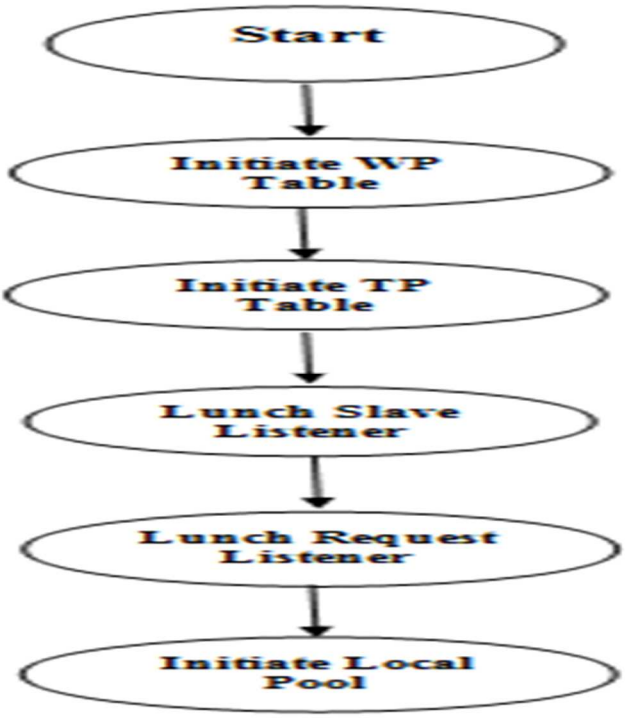

Figure.3: Main Server Algorithm Flowchart

b. Slave Listener Algorithm Flowchart

It starts the slave listener launch for detecting the new slave nodes; if they can't find the new slave node then it can search for it thoroughly. If they can detect the node then it can then it can check that is new are old. If they are not new then it can read the signal and update the stability in TP table and again listen for node. If the node is new then it update the thread pool table and add a new thread pool in TP table and then it can run the thread pool allocator for the allocation of thread on the basis of service times. It lasts the command goes back for detecting the new nodes.

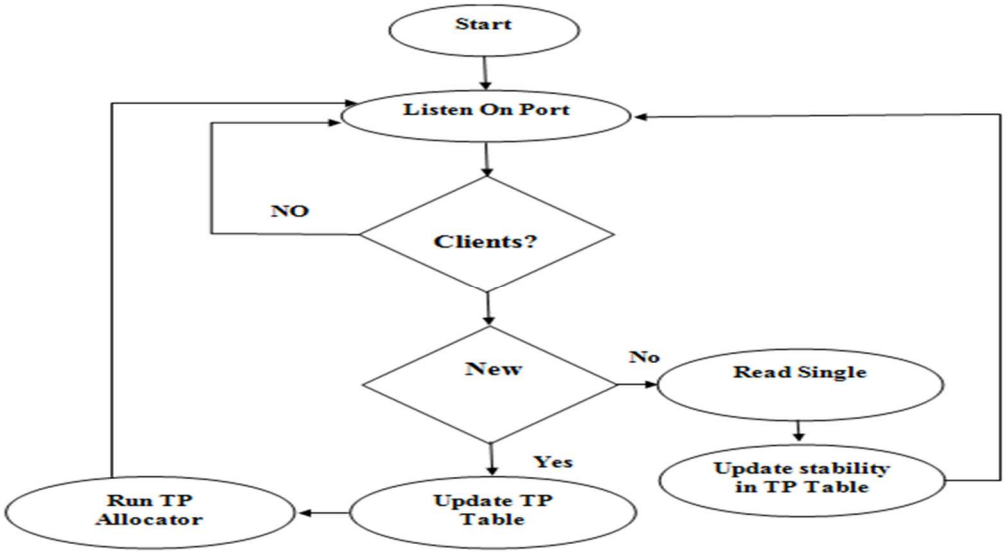

Figure.4: Slave Listener Algorithm Flowchart

c. Request Listener Algorithm Flowchart

Request listener is run to listens the new jobs from the clients thoroughly and then it updates the workload profiling table and passes the job to thread pool allocator for allocation of jobs on the basis of service times.

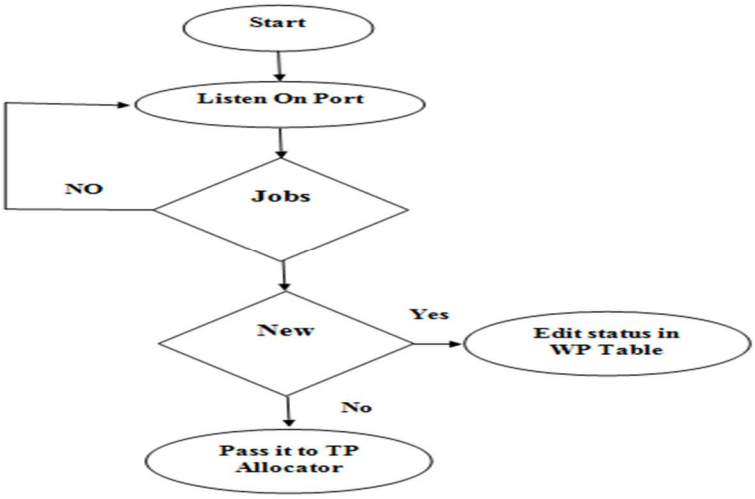

Figure.5: Request Listener Algorithm Flowchart

d. Slave Pool Algorithm Flowchart

At the start of the slave node is connect by giving the IP of the main server. Then it can listens the request from the TP allocator if they cannot detect request it can listens continuously for request if detect the request then they check the stability if stability is no then send stability off to main server. And check the stability continuously until the

node become stable and then send stability on message to the main server and listen for new node. If the node is stable then execute jobs.

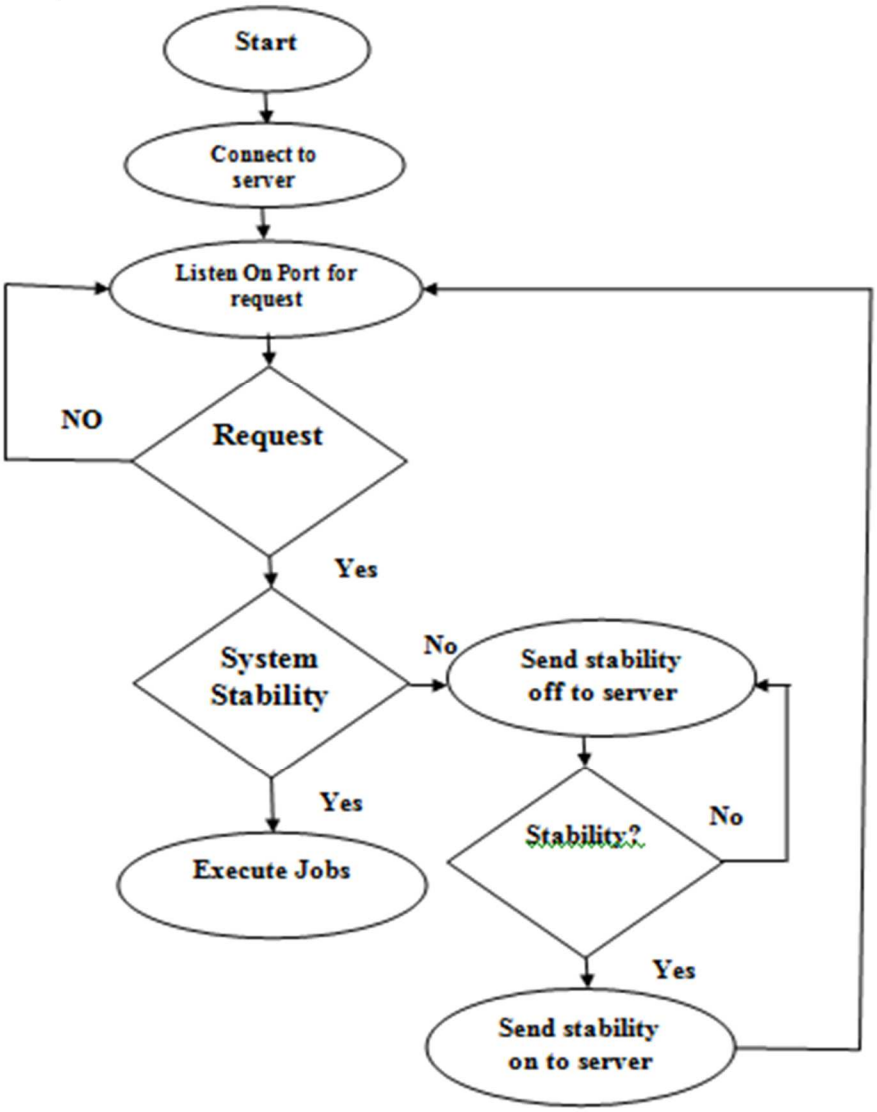

Figure.6: Slave Pool Algorithm Flowchart

e. Thread Pool Allocator Algorithm Flowchart

The main object of our proposed scheme is the thread pool allocator. It starts the TP allocator called by slave listener. If they can detect the new node then they can update the work load profiling table and can also add a new pool in thread pool table, and then it can divide the incoming request from the node by using the formula (Compute n = Active Jobs / Numbers of Pools). If new node is not detected then thread pool allocator is called by request listener if the new incoming jobs are not new means the jobs which are coming from nodes are already stored in work load profiling table. Then it select the job by indexing and then check the stability of the node if the node is not stable show the message that server is busy. If the node is stable then send job to the pool and exit. If the request listener detects the new job then they can distribute it by using the formula (Compute n = Active Jobs / Numbers of Pools).If the work load profiling table has more numbers of jobs then the thread pools, they can select the n number of jobs from WP table and send it to the next pool in TP table. If the entry is null in TP table then it goes back pointer in TP table. If the entry is not null then it checks that all jobs are sorted if yes then exit otherwise they can select the n number of jobs from WP table and can assign it to the next thread pool in thread pool table.

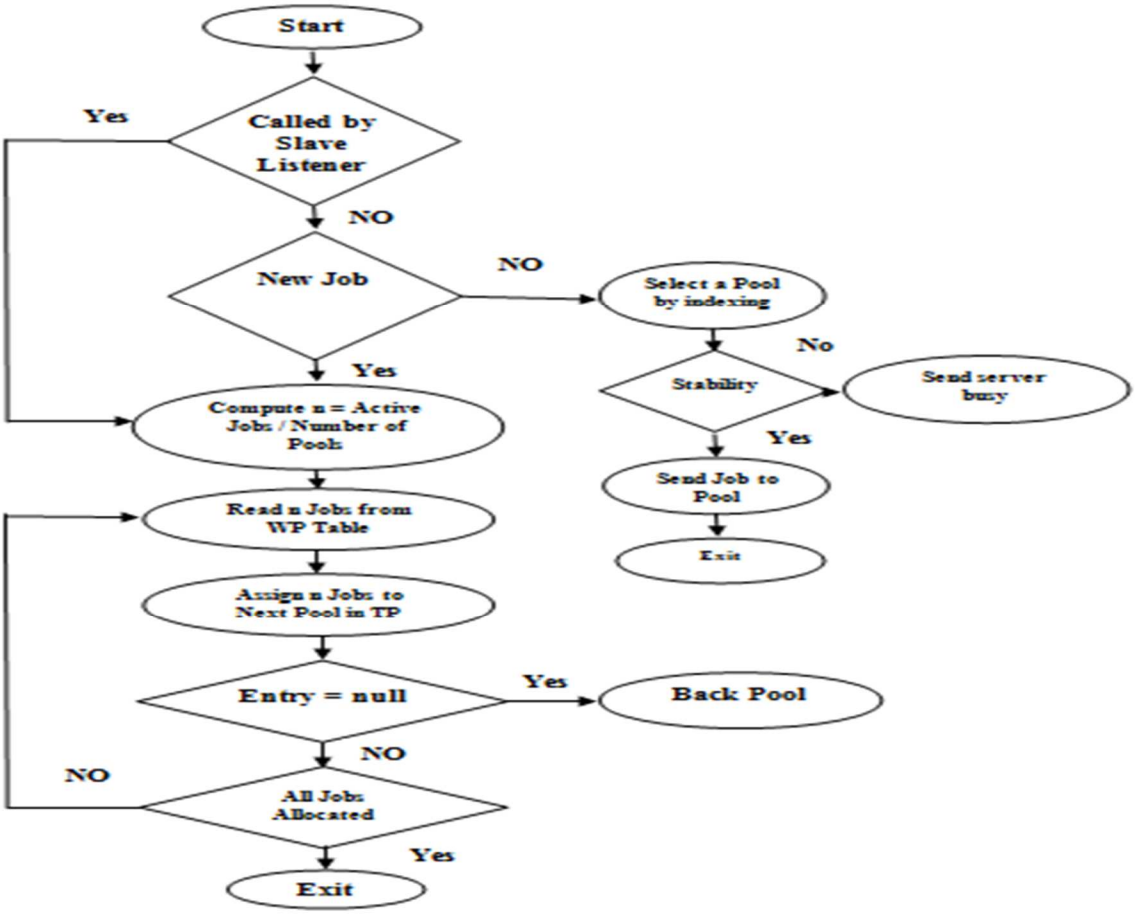

Figure.7: Thread Pool Allocator Algorithm Flowchart

3.4 Performance Metrics

For analyzing our proposed strategy i.e. implementation of multiple thread pools based on distribution of service times with existing strategy DFBOS we use various performance metrics. In this section we will discuss these performance metrics one by one below.

a. Throughput

The number of jobs which can be completed in one second are called throughput.

b. Response Time Of Jobs

The duration from the submission of jobs is to the completion of jobs and provides the required outputs are called the response time of jobs.

c. Thread Pools

Thread pools represent the number of running threads in the pool; in our proposed strategy we use the default thread pool, high service times and low service times thread pools.

d. Wait time

The interval which jobs spent in ready queue, the proposed scheme will minimize the wait time to achieve maximum performance.

4.Analyses

In this section we will discuss about the imitation atmosphere that we have used for validating our proposed thread pool system.

4.1.Thread Pool Tester

We use JAVA based simulator for examining dissimilar requests from clients to achieve the required performance in our proposed scheme. We load the proposed scheme i.e. implementation of multiple thread pools based on distribution of service times for DFBOS on a simulation environment named as Thread – Pool - Tester [17] generally known as TPT. Two agents combine together in Thread Pool Tester that is client and server. On different

machine we run these both agents the machines have Intel core i5 which have four cores. The basic diagrammatical representation of our thread pool tester is shown in figure 8.

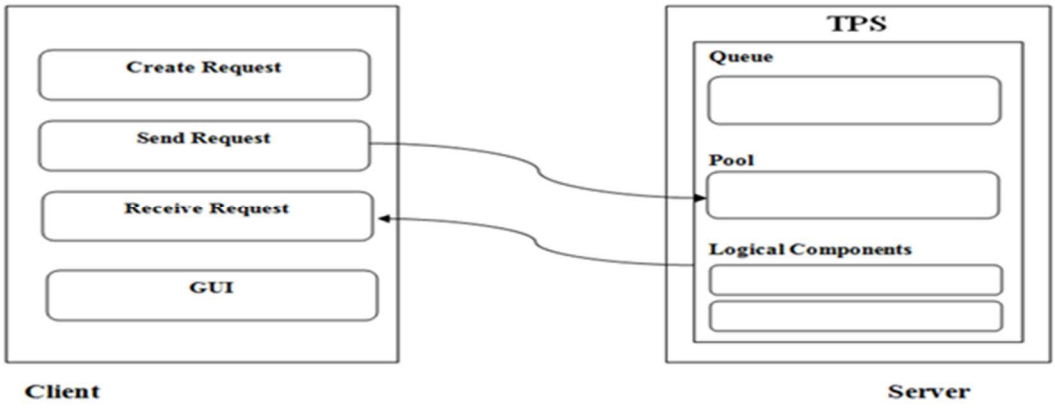

Figure.8: Graphical representation of simulator

a. **Production of Load**

The server has assembled according to the load generation having Poisson division 100λ. We have generated the two types of jobs that are 1kb and 10kb accordingly. Frequencies of each job are 50% that is 50 % for 1kb and 50% for 10kb. Response time of 1kb job is 100ms and 10kb job is 200ms. The cumulative load on the server as shown in figure 9.

4.2. Imitation Atmosphere

For validating our strategy we are performing simulation based environment which consist of three machines, one for server and one for client. We have used Microsoft Window 10 in server machine and Microsoft Window 7 on remaining two systems. The server node is Intel ® Core™ i7 and the remaining two nodes are Intel ® Core™ i5 processors. The main memory of the server node is 8GB and the other two nodes are 4GB. We have performed simulation for one minute according to the load generation having Poisson distribution 100λ. We have generated the two types of jobs that are 1kb and 10kb accordingly. Frequencies of each job are 50% that is 50 % for 1kb and 50% for 10kb. Response time of 1kb job is 100ms and 10kb job is 200ms. The cumulative load on the server as shown in figure 9, we can use the same load generation for existing and proposed strategy and the consequences are plotted from simulation which can be discussed one by one in the next section for analysis.

4.3 Analysis & Results

This sector we will discuss the contrast of existing scheme with proposed strategy. We can compare the proposed scheme that is implementation of multiple thread pools based on distribution of service times with existing strategy DFBOS on the basis of response time and wait time statistics.

Our proposed strategy can divide the thread pools based on distribution of service time which cannot be considered in DFBOS they can only balance the load on nodes. In proposed strategy we can divide the thread pool in low service time and high service time thread pools accordingly. We can use the workload by Poisson distribution that is 100 λ. The client node can send two type of job that is 1kb and 10kb. The response time for 1kb job is 100ms and 10kb job is 200ms. We can store the 100ms job in low service time thread pool and 200ms job in high service time thread pool. After dividing the thread pool on the basis of service times the simulator can generate the graphs for wait time and response time and then we can compare the both strategies.

Figure 9 represents the load generation for both scheme. X – Axis represents time in seconds and Y – Axis represents number of requests. We can use the same load generation of 100 λ. We can perform the load generation for one minute.

Figure.9: Load generation on server

Figure 10 represents the contrast of response time of fulfill task by both strategies.
X - Axis shows the responses/ jobs and the Y - Axis shows the responses time in milliseconds for both strategies. From current graph we can analyze that proposed strategy provide better responses then the existing DFBOS strategy.

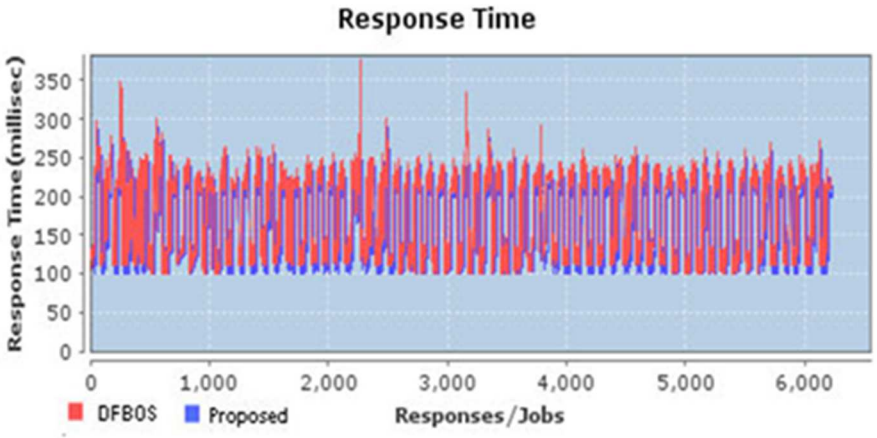

Figure.10: Performance comparisons of response times

Figure 11 represents the comparisons of response time statistics for both strategies in percentile. X - Axis represents the response time in percentile and Y - Axis represents the values in milliseconds. Proposed scheme has increased the response time for 50 percentile is 14% and for 90 percentile is 13% as compare to existing scheme.

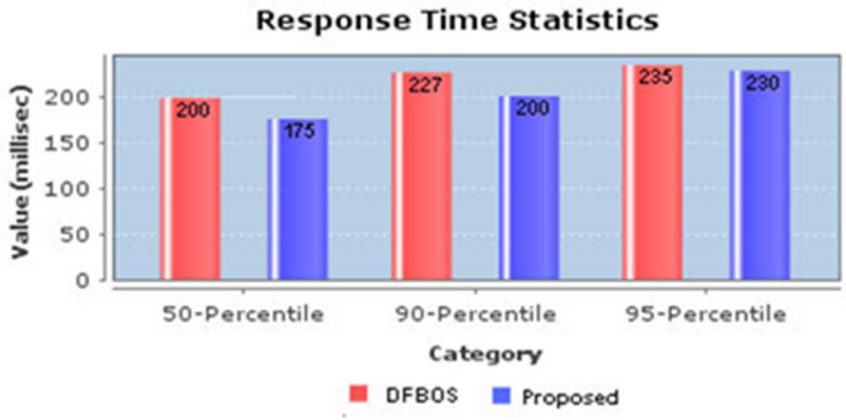

Figure.11: Performance comparisons of response times statistics

Figure 12 represents the comparisons of wait time for both schemes. X-Axis represents the responses / jobs and Y-Axis represents the wait time in milliseconds. Graph shows that proposed scheme has better wait time than existing scheme.

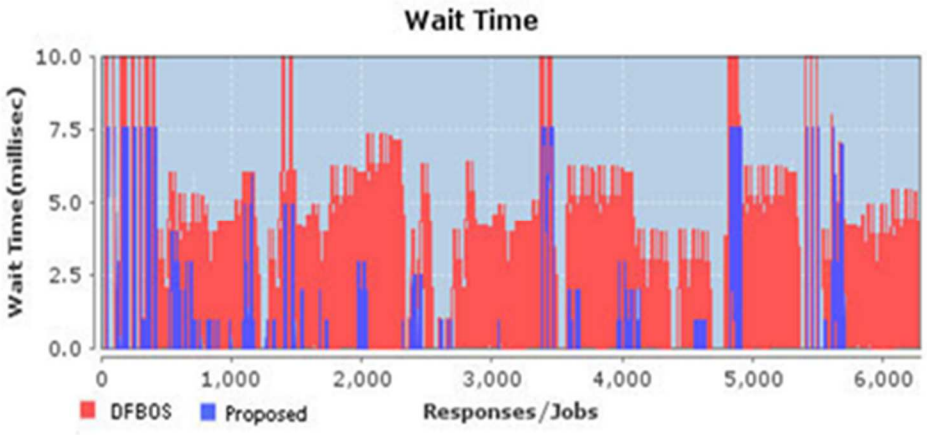

Figure.12: Performance comparisons of wait times

Figure 13 represents the comparisons of wait time statistics for both strategies. X - Axis represents the average wait time and Y - Axis represents the value of wait time in milliseconds. Graph shows that the wait time of proposed strategy is 1 millisecond and the existing scheme is 6 milliseconds.

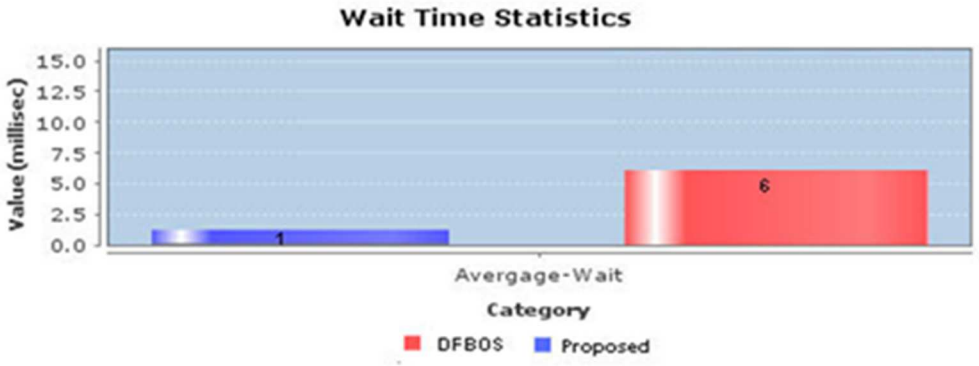

Figure.13: Performance comparison of wait time statistics

5.Conclusion

The proposed strategy can implement multiple thread pool based on distribution of service times for distributed frequency based optimization strategy DFBOS to avoid starvation and achieve concurrency in server site. In our research work we can divide thread pool on the basis of service time into low service time and high service time thread pool in distributed environments.

For comparing both strategies we have used a simulator named as Thread Pool Tester TPT which is a JAVA based simulator and it has shown that proposed strategy is superior then DFBOS.

From our analysis we have concluded that the response time of our research work is worse than the existing DFBOS i.e. for 50 percentile the response time is 14% and for 90 percentile the response time is 13%. The analysis can also conclude that the wait time of proposed strategy is less as compare to the existing DFBOS i.e. wait time of proposed strategy is 1ms and existing scheme is 6ms.

6. Outcome of the Study

- For 50 percentile the response time is 14%.
- For 90 percentile the response time is 13%.
- Wait time of proposed strategy is 1ms and existing scheme is 6ms.
- Avoid starvation for low service time jobs.

7. Future Work

In future we can elaborate our research work to sensor thread level in distributed thread pool environments.

REFERENCES

[1] D. C. Schmidt and S. Vinoski, "Object Interconnections: Comparing Alternative Programming Techniques for Multithreaded Servers - the Thread-Pool Concurrency Model", C++ Report, SIGS, Vol 8, No 4, April 1996.

[2] D. C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers", Communication of the ACM (New York, USA) Volume 41, Issue 10, Oct. 1998, pp. 54-60.

[3] Yue-Shan. Chang, W. Lo, Chii-Jet. Wang, Shyan-Ming. Yuan and D. Liang, "Design and Implementation of Multi-Threaded Object Request Broker". International conference on Parallel and Distributed Systems, (Washington, DC, USA) ISSN : 1521-9097, 1998, pp. 740-747.

[4] S. Hafizah, Ab. Hamid, M. Hairul, N. M. Nasir, W. Y. Ming and H. Hassan, "Improving Response Time of Authorization Process of Credit Card System Using Multi-Threading and Shared-Memory Pool Techniques", Journal of Computer Science 4 (2), ISSN 1549-3636, 2008, pp. 151-160.

[5] D. C. Schmidt and F. Kuhns, "An overview of the real-time CORBA specification," IEEE Computer, vol. 33, no. 6, June 2000, pp. 56-63.

[6] K. Wang, Y. Zhang, Y. Yu, and Y. Li, "Design and optimization of socket mechanism for services in Internet of Things", ;in Proc. WOCC, 2013, pp.327-332.

[7] D. Xu and B. Bode, "Performance Study and Dynamic Optimization Design for Thread Pool Systems", ;in proc. of the Int. Conf. on Computing Communications and Control Technologies. (Austin, Texas, USA), 2004, pp. 167-174.

[8] T. Ogasawara, "Dynamic Thread Count Adaptation for Multiple Services in SMP Environments," IEEE International Conference on Web Services (ICWS '08), Sep 23-26, 2008, pp. 585-592.

[9] N. Chen and P. Lin, "A Dynamic Adjustment Mechanism with Heuristic for Thread Pool in Middleware", 3rd Int. Joint Conf. on Computational Science and Optimization. IEEE Computer Society, (Washington DC, USA), 2010, pp. 324-336.

[10] Y. Ling, T. Mullen, and X. Lin, "Analysis of optimal thread pool size", ACMSIGOPS Operating Systems Review, 34(2), 2000, pp. 42–55.

[11] J.L. Hellerstein, "Configuring resource managers using model fuzzing: A case study of the .NET thread pool" IFIP/IEEE International Symposium on Integrated Network Management (Washington, USA), 2009, pp. 1-8.

[12] J. H. Kim, S. Han, H. Ko and H. Y. Youn, "Prediction- based Dynamic Thread Pool Management of Agent Platform for Ubiquitous Computing", in Proc. of UIC 2007, pp. 1098-1107.

[13] D. Kang, S. Han, S. Yoo and S. Park, "Prediction based Dynamic Thread Pool Scheme for Efficient Resource Usage", ;in Proc. of the IEEE 8th Int. Conf. on Computer and Information Technology Workshop, IEEE Computer Society, (Washington, DC, USA), 2008, pp. 159-164.

[14] Kang-Lyul. Lee, H. N. Pham, Hee-seong. Kim, and H. Y. Youn, "A novel predictive and self-Adaptive Dynamic Thread Pool management", Ninth IEEE International Symposium on Parallel and Distributed Processing with applications, (Busan, Korea), 26-28 May, 2011, pp. 26-28.

[15] S. Ramisetti and R. Wanker, "Design of hierarchical Thread Pool Executor", Second International conference on modeling and Simulation, (kualalampur, Malaysia), 2011, pp. 284-288.

[16] A. Bhattacharya, "Mobile Agent Based Elastic Executor Service", Ninth International Joint Conference on Computer Science and Software Engineering (JCSSE), 2012, pp. 351-356.

[17] F. Bahadur, M. Naeem, M. Javed, A. Wahab, "FBOS: Frequency Based Optimization Strategy For Thread Pool System", The Nucleus 51, No. 1, 2014, pp. 93-107.

[18] Sheraz Ahmad "Load Balancing in Distributed Framework for Frequency Based Thread Pools"/thesis IT Department Hazara University Mansehra

[19] F. Bahadur, "Thread Pool Tester Simulation Tool" Available: https://github.com/faisalsher/ThreadPoolTester, Aug. 12, 2015 [Accessed: Aug. 12, 2015].

LIST OF ABBREVIATIONS

DFBOS	Distributed Frequency Based Optimization Strategy
TPT	Thread Pool Tester
AIT	Average Idle Time
COM	Common Object Model
CORBA	Common Object Requested Broker Architecture
DCOM	Distributed Common Object Model
DOC	Distributed Object Computing
FBOS	Frequency Based Optimization Strategy
FIFO	First In First Out
IPC	Inter Process Communication
OMG	Object Management Group
ORB	Object Request Broker
QOS	Quality of Service
RT	Real Time
SMP	Symmetric Multi-Processing
TAT	Turn Around Time
IOT	Internet of Things
CAS	Code Arrangement for Shared Resources
AIT	Average Idle Time
TEMA	Tendency Exponential Moving Average System
WP	Work Load Profiling
TP	Thread Pool
MS	Millisecond