Hybrid Synchronization Based Distributed Thread Pool

Faisal Bahadur, Arif Iqbal Umar, Fahad Khurshid, Ali Imran Jehangiri, Ibrar Afzal

Department of Information Technology, Hazara University, Mansehra, K.P.K. Pakistan

Summary
Distributed applications have been frequently developed with distributed thread pools (DTP) for performance boost. An important design aspect of thread pool system (TPS) is the use of one of two synchronization mechanisms called mutex and spinlock that controls access to producer-consumer shared queue (PCSQ). When to use which one depends upon several factors including computer architecture and application behavior. Each one has its own pros and cons. In our previous work, we have proposed a distributed frequency based thread pool (DFBTP), where PCSQ was implemented by mutex. Mutex causes context switches due to sleeping and waking of threads, hence degrades system performance. In this paper we are presenting a new distributed thread pool named Hybrid Synchronization Based Distributed Thread Pool (HSBDTP) that implements a combined synchronization approach of both spinlock and mutex in order to gain advantages of both synchronization primitives. The evaluation results have proved that HSBDTP outperformed old approach by sustaining maximum performance in terms of response time and wait times.

Key words:
Distributed System, Multi-threading, Thread Pool System, Performance.

1. Introduction

The ever-growing expansion of internet and World Wide Web demands scalable services that must be performance efficient and highly available. The prominent progression of internet's user doubles internet traffics every two or three months. For example, OSN sites such as LinkedIn, Flickr, Myspace, Twitter and Facebook provide facilities to over half a billion users at the same time [1]. The OSN's not only provide basic communication capabilities but also provide other services by third party applications e.g. sharing documents, sending virtual gifts, or gaming. Facebook alone is running over 81,000 third-party applications [1]. These third-party applications have a profound impact on the application server's scalability and performance thus results in additional traffic. For example, when Facebook launched its developer platform, the traffic increased by 30% in a week after launching [2], while in case of Twitter the traffic increased by a factor of twenty after opening up its API [3]. Also, the variations in demand go to extreme levels in some
internet services that cause overload condition and needs special attention to manage server side resources. In order to deal with these complexities, internet services are provided by distributed application servers that are responsible of providing run time services to one or more applications, where these applications service requests to a large number of concurrent users.

Today, distributed systems have been implemented in almost all domains including telecommunication, defense, industrial automation, financial services, entertainment, government and e-commerce. And that is why, the requirements of complexity management, scaling and overload management are increasing day by day. As discussed earlier, distributed systems handle heavy workloads, where client's requests are incoming from a remote source through some network protocol. These heavy workloads are handled by distributed systems through extremely concurrent design configurations that are implemented as middleware. The performance of distributed systems is dominated by middleware that provide different functionalities, e.g. multithreading policies, remote communication mechanisms, persistence services and transaction management etc. [4]. It is the middleware that makes distributed system scalable, highly available and highly performant [4]. Some remarkable examples of middleware services for distributed systems are Distributed Object Computing (DOC) middleware (such as CORBA, SOAP, RMI) Component middleware (such as .NET, Java Beans, CORBA Component Model), Message Oriented Middleware (such as Java Message Queue, BEA's WebLogic MessageQ) etc.

One of the most important performance related feature of any middleware service in distributed systems is concurrency control that handles multiple concurrent requests. Two most commonly used concurrency models are Thread Pool System (TPS) and event driven model (EDM).

As compared to TPS, EDM is more performance efficient, but at the same time it is much complicated and challenging to implement than TPS [5]. The most challenging task in EDM is to handle scheduling and assembling of events [6]. Moreover, EDM leads to enormous cascading callback chains [7]. As compared to EDM, TPS offers more solid structuring constructs for concurrent servers by means of threads that are light weight and represent work from the perception of the task itself [8,9]. Moreover, TPS avoids resource thrashing and overheads of thread creation and destruction [10]. Some examples of TPS in middleware for

Manuscript received July 5, 2018
Manuscript revised July 20, 2018

distributed systems include .NET thread pool [11], Java Message Queue Thread Pool [12].
A typical TPS contains a request queue, a pool of threads (workers) and dynamic optimization algorithm (DOA) that optimizes pool size, as shown in Figure (1).

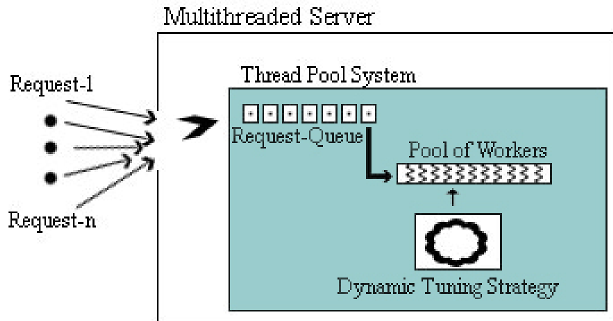

Fig. 1 Conceptual model of thread pool system embedded in a server

Request queue stores incoming client's requests. Worker threads in the pool fetches and executes these requests. These worker threads in the pool are recycled (instead of being destroy) to process next client's request from queue. The re-spawning and recycling of worker threads avoids thread creation and destruction costs, but, under a heavy load scenario, additional threads must be dynamically created and inserted inside pool to cope with the load. The DOA component of TPS is responsible to decide the quantity of extra threads. It is a challenging task of DOA to maintain an optimal pool size on run time, in order to produce better response times and maximum throughput so that quality of service can be maintained. If thread pool size is beyond an optimal limit, then it increases thread context switches and thread contention (on shared resource), that ultimately provides poor performance. On the other hand, if pool size is smaller than an optimal limit then it results in poor response time and throughput. Handling this tradeoff on run time is essential to achieve best performance. Optimizing thread pool size by DOA is not an exact science and it can be performed on the basis number of parameters and factors.
The variety of target servers where TPSs are installed makes DOA more challenging, as there are varied characteristics of the deployment system with a diverse nature of tasks. Because of this reason, TPS has been evolved from single-pool to multi-pool and from multi-pool to distributed. Distributed thread pools (DTP) are designed for distributed systems where they are horizontally scaled over number of nodes available on the network. Each node has its own TPS and a central server forwards requests to these nodes as shown in Figure (2).

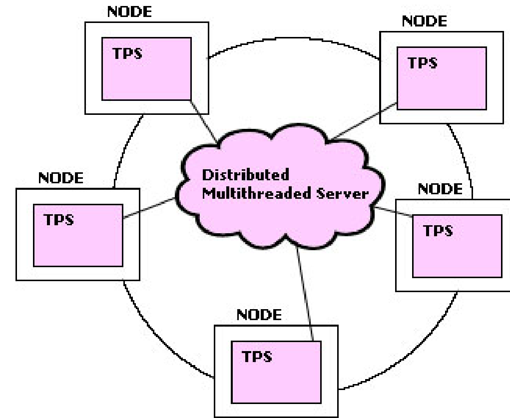

Fig. 2 Conceptual model of distributed thread pool system

We have presented DFBTP in our previous work [13], where an explicit instance of TPS is running on each slave node. Each TPS is optimized by request rate on corresponding node. In DFBTP, we used mutex based PCSQ at server and slave nodes. In case of mutex based PCSQ only one thread can acquire a lock, putting all other threads to sleep. The winner thread dequeues a request and then unlock PCSQ for other threads to pick a request. All other threads wakeup and try to acquire a lock on PCSQ. Frequently putting threads to sleep and wake states is expensive in terms of time as it needs a lot of CPU instructions. If mutex is locked for very short amount of time, then the time spent in sleeping and waking of a thread might exceed the time a thread has actually slept by far. With plenty of short time locks, the time wasted for continuously putting threads to sleep and wake decreases runtime performance. Another disadvantage of mutexes is that the sleeping threads cannot do any useful work even some CPU-cores are free to use.
An alternative of mutex based queue is spinlock based queue that do not let a thread to sleep (if the lock is already held by another thread), instead, the looser thread takes benefit of its full runtime quantum and tries again to acquire a lock on PCSQ. It then immediately continues its work by acquiring the lock on PCSQ. But at the same time, spinlock persistently wastes CPU time and if a lock is held longer, then this will waste a lot of CPU time and it would have been better if the thread was sleeping instead.
In case of PCSQ, the lock is held for short amount of time as it just requires to enqueue or dequeuer operations, so using some spinlocks (by threads) on PCSQ can greatly improve performance.
In order to achieve advantages of both synchronization primitives, this paper presents a hybrid synchronization based distributed thread pool (HSBDTP) that utilizes combination of both mutex and spinlock that avoids context switch overhead by spinlock and CPU-cycles overhead by mutex.

The rest of the paper is organized as follows. Section 2 presents previous work. Material and methods are presented in section 3. Proposed system is validated through simulation in section 4 and finally conclusion is given in section 5.

2. Related Work

An investigation of optimal pool size is presented in [14] that developed a mathematical model that is based on a relationship among system's request rate, pool size, thread context switch and costs associated with thread creation and destruction. However, it is difficult to precisely estimate the time for thread context switch and thread creation and destruction. Therefore, estimated pool size might be erroneous.

Idle time of requests that are waiting in the request queue are utilized in [15] in order to dynamically optimize pool size. An average idle time of all requests waiting in the queue is calculated after threshold time period and compared with the previous average idle time. Pool size increases by certain threshold value if average increases otherwise pool size is reduced.

TPS presented in [16] used prediction scheme by Gaussian distribution in order to predict the pool size in advance. Due to synchronization overhead these predictions may be inaccurate.

Another prediction based TPS is presented in [17] that calculated exponential moving averages of change in pool size. Redundant threads were created by this TPS if predictions are wrong.

Multiple pools of thread were utilized in [18] where each pool servers specific service, and thread borrowing scheme between pools was used. But, managing multiple pools was itself a cost effective procedure.

TPS presented in [19] used a model fuzzing approach in order to optimize pool size. Number of constraints and parameters were applied for dynamic optimization which was too difficult to rapidly make a decision, hence this scheme was not suitable for the system having frequent change in request rates.

Response coefficients were calculated in [20] for dynamic optimization of TPS. But, response coefficient is normally effected by number of run time parameters.

TPS developed in [21] was extension of [17]. This TPS analyzed trends of time series in order to avoid redundant threads. But its downside was creation of lacking threads.

A speculative framework for distributed TPS was presented in [22], that was governed by software agents that optimized pool size on the basis of load conditions. But, no prototypical verification was presented.

A framework of hierarchical thread pool executor was presented in [23] that was targeted to only DSM systems.

A multiple request queue scheme is utilized in [24] where a single pool of threads serves multiple request queues and each queue stores particular type of requests. URL is used to classify type of request. Each request arrived at the server is pushed to its corresponding queue by using a lookup table. Each queue is allotted specific number of threads in the pool based on the average service time of waiting requests and request arrival rate.

TPS presented in [25] applied a divide and conquer approach that divide a task into subtasks and run those tasks in parallel in order to reduce computational cost. But, it was very difficult to dynamically divide a task into subtasks.

A multiple pool approach is used in [26] where each pool is reserved for requests having specific service time. In this way requests having large service times are separated from requests having small service times, hence avoiding large requests to block small ones in order to occupy all threads in the pool.

In our previous work [13], we have presented a DFBTP with mutex based PCSQ at server and slave nodes, where an explicit instance of TPS is running on each slave node. Each TPS is optimized by request rate on corresponding node. Due to mutex based PCSQ, DFBTP faces overheads of context switches and thread contention.

3. Material and Methods

3.1 Hybrid Synchronization Scheme

The PCSQs used in the proposed DTP utilizes hybrid synchronization scheme. The synchronization strategy proceeds by behaving spinlock first. If PCSQ is not locked by a thread, the thread won't be put to sleep (to avoid context switch overhead), instead, it will perform two more attempts. In case it fails in successive attempts, it changes its strategy to mutex and go to sleep for 10 milliseconds. After waking up it again tries to acquire a lock. Figure (3) shows the strategy of a Consumer thread that dequeues request from PCSQ. If a Consumer thread succeeds in acquiring a lock on PCSQ in any attempt, then it will dequeue request (if queue is not empty) and unlock PCSQ, execute request and again proceed by spinlock strategy. If PCSQ is empty, then thread will go to sleep for 10 milliseconds and again try locking PCSQ after waking up.

Figure (4) shows the strategy of a Producer thread that enqueue request in PCSQ. If a Producer thread succeeds in acquiring a lock on PCSQ in any attempt, then it will enqueue request and unlock PCSQ. Later, when it wants to produce again it will proceed by the same spinlock strategy. If the lock is not acquired till three attempts, then producer thread will back off and go to sleep for 10 milliseconds. It will again proceed by same strategy after waking up.

This scheme neither lets a thread to monopolize the CPU core (and gives a chance to other thread to get CPU

resource), nor this scheme causes too many context switches (as it might let a thread to grab PCSQ in its

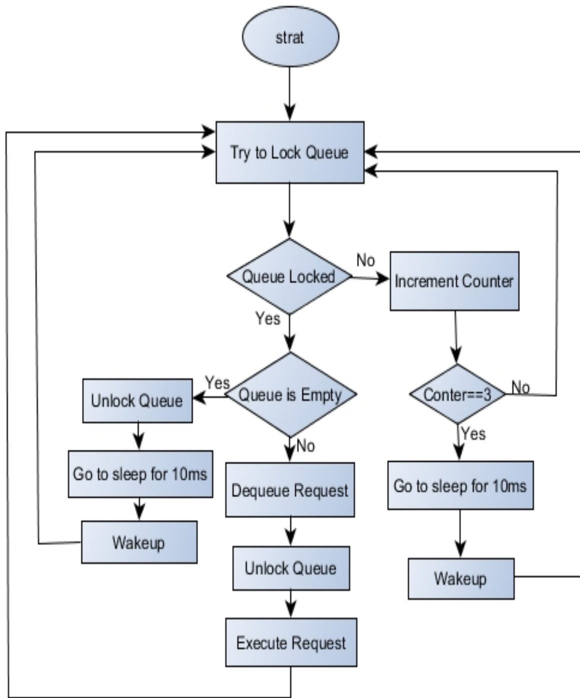

Fig. 3 Consumer's Hybrid Synchronization Scheme

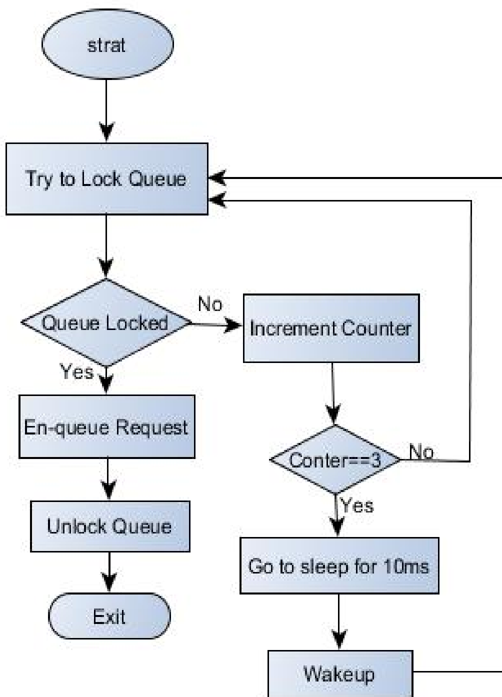

Fig. 4 Producer's Hybrid Synchronization Scheme

successive three attempts). The hybrid synchronization scheme behaves smoothly when there are too many threads in the pool trying to conquer PCSQ, as it generates a mixture of spinlock and mutex scenario, having benefits of both strategies. In this way the hybrid model neither causes wasting CPU cycles in sleeping and waking of threads, nor it allows CPU monopolization.

3.2 Proposed Distributed TPS Architecture

The architecture of proposed distributed TPS is shown in Figure (5).

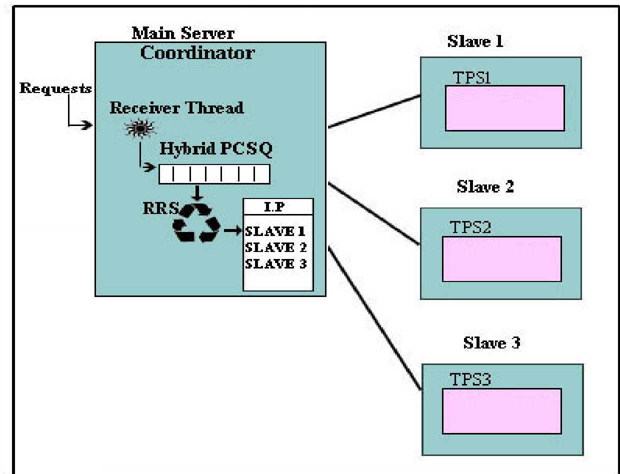

Fig. 5 Proposed Distributed Thread Pool Architecture

The Coordinator is a central authority that starts first. It first initializes RRS, a hybrid PCSQ and a linked list (to hold IP's of slave systems where TPSs are running). The hybrid PCSQ initialized by the Coordinator is a Single-Producer-Single-Consumer queue. It is accessed by a Receiver thread and RRS thread. A Receiver thread will receive client's request and put it in the PCSQ, that is picked by RSS (consumer thread) that forwards it to TPS. All requests arrived at Coordinator are sent to available TPS by RRS that loops on the linked list that contains IPs of available TPSs. RSS and Receiver Thread will apply hybrid synchronization scheme on PCSQ.

3.3 Proposed TPS Architecture

The Architecture of proposed TPS is shown in Figure (6). When a TPS starts on a slave server it initializes its system components that governs TPS execution. First of all, two hybrid PCSQs (Request Queue and Response Queue) are dynamically initialized. After initializing queues, TPS connects to the Coordinator with the help of Connection Manager. Coordinator stores its corresponding IP in the linked list. Next, TPS waits for client's requests (sent by RRS from server side). On request arrival, the value of

Counter is incremented and the request is stored in request queue that is picked by a thread resides in the thread pool. Request Queue is a Single-Producer-Multiple-Consumer queue. A single receiver thread will receive client's requests (sent by RRS) and store these requests in this queue, whereas worker threads in the pool will try to pick these requests by acquiring a hybrid lock on PCSQ. Response queue is a Multiple-Producer-Single-Consumer queue. Threads in the pool are responsible to put processed requests in this queue (in form of response) and a single consumer thread will dequeue each response from the queue and send it to Coordinator that will in turn send it to the client.

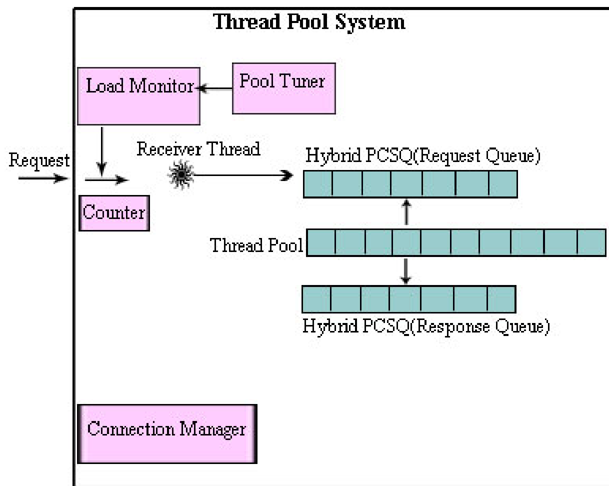

Fig. 6 Proposed TPS Architecture

The Load Monitor is a timer object that monitors client' request rate by repeatedly activating after every second. It reads the value of Counter and again sets it to zero so that next request arrival rate can be measured. Pool Tuner also activates after every second, reads current request arrival rate from load monitor and compares it with the previous rate. If rate is increasing, then Pool Tuner will increase thread pool size (by adding more threads) and makes pool size equal to the request rate. In order to shrink pool size each thread has an internal timer object that starts when a thread becomes free. If a thread is free till five seconds in the pool, then timer object will delete its thread. On low requests rates some threads in the pool would be free that will be deleted automatically in order to reduce pool size.

4. Results and Discussion

In this section we are validating the performance of proposed system by a JPoolRunner simulation kit. JPoolRunner is a client-server based system. We first loaded Coordinator at server tier of JPoolRunner. The client-side of toolkit consists of load generator and GUI system that plots simulation results.

Testing is done on a network of four computers. One computer is used for client-side of JPoolRunner, second computer is used as a main server where we ran Coordinator that accepts requests sent by the load generator. Remaining two machines are slave servers, each is running a TPS.

Table (1) lists testing parameters. JPoolRunner simulates client's requests by set of Task objects with different service times. In this testing we have used a single Task of 100ms. Load generator of JPoolRunner will burst set of these tasks to coordinator for processing. We have selected poisson load generation strategy with an average request frequency of 1000 (requests).

Table 1: Test Plan Parameters

Workload	***Service Time***	***Load Generation***	***λ***
Task	≈100millisec	Poisson	1000

We performed testing for 60 seconds. Figure (7) shows dynamic request rate that is generated by load generator of JPoolRunner, that sent this load of Task objects to the main server where Coordinator is running. In Figure (7), x axis shows time in seconds, whereas y axis shows the load sent on specific time. The load in Figure (7) shows that request rate (1000 frequency) is generated by poisson distribution. There are almost sixty-thousand requests submitted to the server in 60 seconds.

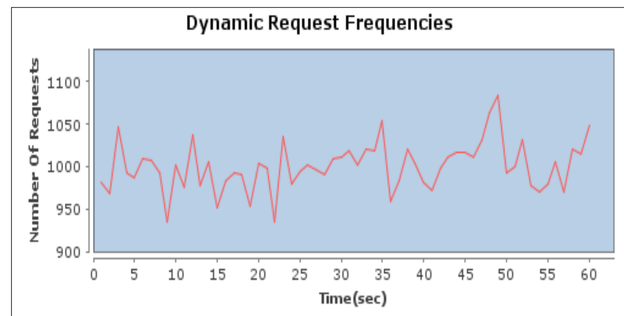

Fig. 7 Load generated for simulation with Poisson Distribution

Figure (8) presents a comparative analysis of response time between proposed scheme(HSBDTP) and old scheme(DFBTP), where x-axis shows each response received at client-tier of JPoolRunner and y-axis shows response time of each response in milliseconds. HSBDTP has significantly minimized response times as compared to DFBTP, because hybrid synchronization scheme in HSBDTP produces less context switch overhead as it utilizes spinlocks most of the time that avoids context switches, whereas in DFBTP the time is wasted in frequently putting threads to sleep and wake states that ultimately decreased runtime performance.

Fig.8 Comparative Analysis of Response Times

In DFBTP, plenty of locks are held for a very short amount of time, threads contention on the shared resource (Request Queue) is high, due to which, requests spent more time in waiting queue (Request Queue) that can be seen in Figure (9) that represents a comparative analysis of wait times of requests in request queue between DFBTP and HSBDTP. The x-axis shows each response received and y-axis shows wait time of each response in milliseconds. HSBDTP outperformed DFBTP by minimizing wait times of requests noticeably, as mutex based PCSQ (Request Queue) in DFBTP is causing so many context switches and run time overhead, hence the process of picking requests from queue is slow that increased their wait time in the queue. In DFBTP, many threads are competing to lock PCSQ and only one will acquire lock that causes all other thread to withdraw and go to waiting state by performing context switches. In case of HSBDTP the overhead of thread contention on hybrid PCSQ (Request Queue) is minimal due to spinlock and requests are picked by threads quickly, hence reduced wait times.

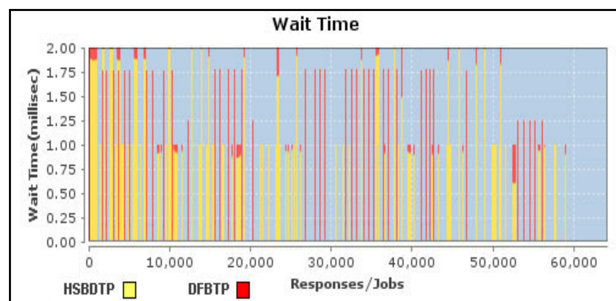

Fig. 9 Comparative Analysis of Wait Times

5. Conclusion

This work presented a distributed thread pool that utilized a hybrid synchronization scheme by using combination of spinlock and mutex in order to gain advantages of both primitives. The simulation results have proved that hybrid synchronization based distributed thread pool system is more performance efficient in terms of response and wait times than mutex based distributed thread pool, because hybrid synchronization scheme produces less context switch overhead as it utilizes spinlocks most of the time that avoids context switches, whereas in DFBTP the time is wasted in frequently putting threads to sleep and wake states, that ultimately decreased runtime performance.

Acknowledgment

We are grateful to Hazara University, Mansehra, Pakistan for their support to publish this research work.

References

[1] A. Nazir, S. Raza, D. Gupta, C-N. Chuah, B. Krishnamurthy. Network level footprints of facebook applications. In Proceedings of the 9th ACM SIGCOMM conference on Internet measurement, 2009, pp. 63-75. ACM.
[2] A. Nazir, S. Raza, and C.-N. Chuah. Unveiling facebook: A measurement study of social network based applications. In Proc. Internet Measurement Conference (IMC), 2008.
[3] B. Krishnamurthy, P. Gill, and M. Arlitt. A few chirps about twitter. In Workshop on Online Social Networks, 2008.
[4] G. Denaro, A. Polini, W. Emmerich. Performance testing of distributed component architectures. In Testing Commercial-off-the-Shelf Components and Systems. Springer, Berlin, Heidelberg, 2005, pp. 293-314.
[5] M. Andreolini; V. Cardellini; M. Colajanni. Benchmarking models and tools for distributed web-server systems. In: IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation, September, 2002, pp. 208-235.
[6] M. Welsh; D. Culler; E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. ACM SIGOPS Operating Systems Review, 2001, 35(5) pp. 230-243.
[7] A. Gustaffson. Threads without the Pain. Queue, 2005, 3(9):34-41.
[8] R.V. Behren; J. Condit; E. Brewer. Why events are a bad idea (for high-concurrency servers). In: Proceedings of the 9th conference on Hot Topics in Operating Systems, May, 2003, pp.4-4.
[9] R.V. Behren; J. Condit; F. Zhou; G.C. Necula; E. Brewer. Capriccio: scalable threads for internet services. ACM SIGOPS Operating Systems Review, 2003, 37(5): 268-281.
[10] T. Peierls; B. Goetz; J. Bloch; J. Bowbeer; D. Lea; D. Holmes. Java Concurrency in Practice, Addison-Wesley Professional, 2005.
[11] The Managed Thread Pool.
[12] https://msdn.microsoft.com/enus/library/0ka9477y(v=vs.110).aspx (accessed on 10 July 2018).
[13] ORACLE, Java System Message Queue 4.3 Technical Overview
[14] https://docs.oracle.com/cd/E19316-01/820-6424/aerck/index.html (accessed on 10 July 2018).
[15] S. Ahmad, F. Bahadur, F. Kanwal, R. Shah, “Load balancing in distributed framework for frequency based thread pools,” Computational Ecology and Software,2016, Vol. 6, No. 4, pp. 150-164.

[16] Y. Ling, T. Mullen, X. Lin, “Analysis of optimal thread pool size,” ACM SIGOPS Operating System Review,2000, Vol. 34, No. 2, pp. 42–55, 2000.
[17] D. Xu, B. Bode, “Performance Study and Dynamic Optimization Design for Thread Pool System,” In Proc. of the Int. Conf. on Computing Communications and Control Technologies, Texas, USA, 2004, pp.167-174.
[18] J. Kim, S. Han, H. Ko, H. Youn, “Prediction- based Dynamic Thread Pool Management of Agent Platform for Ubiquitous Computing,” International Conference on Ubiquitous Intelligence and Computing, Springer, Berlin, Heidelberg, 2007, pp. 1098-1107.
[19] D. Kang, S. Han, S. Yoo, 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.
[20] T. Ogasawara, "Dynamic Thread Count Adaptation for Multiple Services in SMP Environments," IEEE International Conference on Web Services (ICWS '08), Beijing, China,2008, pp. 585-592, September.
[21] J. Hellerstein, “Configuring resource managers using model fuzzing: A case study of the .NET thread pool,” IFIP/IEEE International Symposium on Integrated Network Management (IM '09), Long Island, NY, USA, 2009, pp. 1-8.
[22] N. Chen, 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.
[23] K. Lee, H. Pham, H. Kim, H. Youn, O. Song, “A novel predictive and self-adaptive dynamic thread pool management,” In: Proceedings - 9th IEEE International Symposium on Parallel and Distributed Processing with Applications, Busan, South Korea, 2011, pp. 93–98.
[24] P. Martin, A. Brown, W. Powley, J. Luis, V. Poletti, “Autonomic management of elastic services in the cloud,” In: Proceedings - IEEE Symposium on Computers and Communications, Kerkyra, Greece, 2011, pp. 135–140.
[25] S. Ramisetti, R. Wankar, “Design of hierarchical thread pool executor for DSM,” Second International Conference on Intelligent Systems Modelling and Simulation (ISMS), Kuala Lumpur, Malaysia, 2011, pp. 284-288.
[26] G. You, Y. Zhao, “A weighted-fair-queuing (WFQ)-based dynamic request scheduling approach in a multi-core system,” Future Generation Computer System, Vol. 28, No.7, 2012, pp.1110–1120.
[27] M. Chen, Y. Lu, G. Zhang, W. Zou, “Real-Time Simulation in Dynamic Electromagnetic Scenes Using Pipeline Thread Pool,” Tenth International Conference on Computational Intelligence and Security (CIS), Kunming, China, 2014, pp. 770-774.
[28] J. Mace, P. Bodikz, M. Musuvathiz, R. Fonseca, K. Varadarajanz, “2DFQ: Two- Dimensional Fair Queueing for Multi-Tenant Cloud Services,” In Proceedings of the ACM SIGCOMM Conference, ACM, Florianopolis, Brazil, 2016, pp. 144-159.