2013

Illinois Institute of Technology, Chicago

Anusree Kailash

[DISTRIBUTED JOB SCHEDULING FRAMEWORK]

This project is a study, implementation and analysis of a distributed job scheduling framework meant to run jobs on a distributed system with good scaling, load balancing and fault tolerance characteristics.

PROJECT REPORT

Distributed Job Scheduling Framework: Study, Design and Analysis

TABLE OF CONTENTS

Introduction or Motivations

In modern high performance computing, the use of large scale distributed systems ensure that job allocation and resource management with an eye on efficient load-balancing is key criteria in determining the scaling performance of the system.

A job can be expressed as one or more independent tasks which wait for input to be available, perform computations and produce output. Systems such as DAGMan [XII], Karajan[XIII], Swift[XIV][XV], and VDS[XV] support this model. These systems have all been used to encode and execute thousands of individual tasks. Many of these tasks are executable at once and a parallel system can be used to execute these tasks in parallel to achieve high throughput.

Currently there are some task scheduling algorithms written which are centralized(Eg: Falkon[V], Sparrow[XXI]) which are found to be not scaling well[VII]. The decentralized algorithms also often rely on certain centralized functions to reduce the complexity of the system. In such cases, a comparison can be made and new approaches may be arrived at to mitigate the complexity.

Falkon[V] uses a central job dispatcher which eliminates the need for supporting complicated functions in the batch scheduler. For this project, instead of a central job dispatcher or a complicated batch scheduler (as seen in older systems), I explore the possibility of decentralized pull based model, where the processors which are idle, picking up jobs from a distributed queue and executing them.

To implement the common job queue in a distributed fashion, I explored the available Distribute Hash Tables(DHT) implementations. There are many implementations of DHT in place today which allow the shared queue of jobs between nodes. Fault tolerance is also to be incorporated to prevent jobs from being lost due to node crash.

Some of the DHT considered for this project is Cassandra, Chord[I] and Hazelcast[X]. Chord was last updated 6 years ago and is still in experimental state whereas Hazelcast has a stronger developer and user community which makes the case for selecting Hazelcast stronger. Cassandra is an eventually consistent persistent, high availability data store, which is good for write-heavy, read-low type of systems. Since my system requires a strictly consistent, very fast distributed queue implementation which supports high reads and high writes, Cassandra does not fit my purpose. Hazelcast has a concurrent queue implementation which is stored in distributed RAM, which fits my requirement of the distributed job queue well. Hazelcast is selected for DJSF implementation on the basis of the ease of use and its ability to support various configurations with minimal changes in the configuration XML file. It is truly distributed and able to handle node failures and redistribute data among the remaining nodes.

For performance analysis of the DJSF, I use the Amazon EC2 cluster which offers Infrastructure-as-a-service. The model is chosen for ease of use and cost-effectiveness. CloudKon[VII] also uses this infrastructure. The nodes required for my distributed system can be rented by the hour and it offers a range of different types of instances with compute/memory/network capabilities of my choosing. An instance is a running virtual machine on Amazon's cloud platform. Each of these instances is deployed with an Amazon Machine Image (AMI). I can create an AMI of my choosing by selecting the operating system to be installed in it and installing the application software I need(Eg: Hazelcast, JDK) and my own DJSF program(client/executor/cluster-node). Users can launch one or more instances of the same AMI by specifying the instance type. Then the instances will be deployed on the cloud and user can connect to them via SSH using their public IP address.

The goal of the project is to study, implement and understand the challenges in job scheduling, scaling and fault-tolerance related issues for a distributed system. I also intend to come up with solution ideas and conclusion based on the analytical data. I do not implement a resource managing algorithm here. Fully featured resource managers such as Condor [XII][XVII], Portable Batch System (PBS)[XVIII], Load Sharing Facility (LSF)[XIX], and SGE[XX] support client specification of resource requirements, data staging, process migration, check-pointing, accounting, and daemon fault recovery. Currently, my DJSF concentrates on task allocation and feedback without any central component in place.

LIST OF PROJECT CONTRIBUTIONS

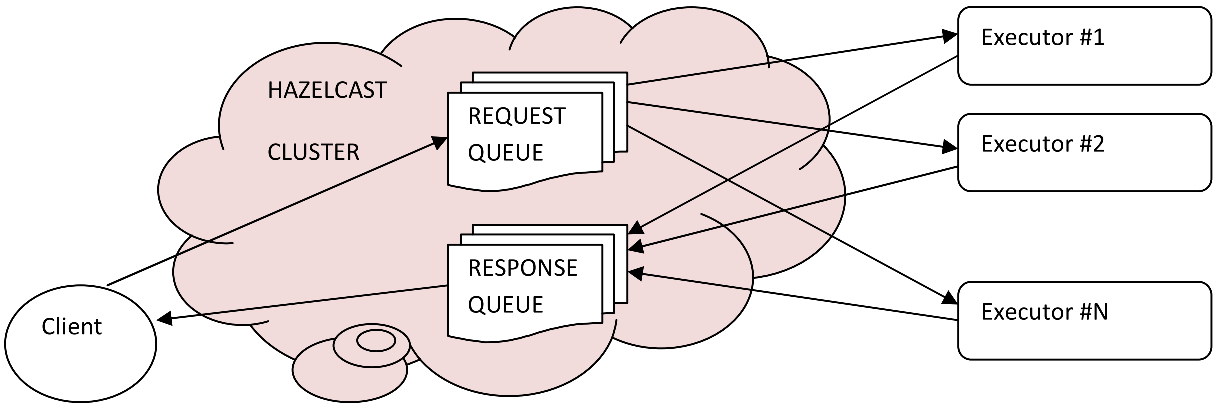
This is an individual project. The study, implementation and analysis as well as the report is made by me alone. My name and CWID for easy reference:

Name: Anusree Kailash

CWID: A2026020

RESEARCH DESCRIPTION

DJSF ARCHITECTURE



The above diagram shows the basic architecture of the DJSF -Distributed Job Scheduling Framework. There is the flexibility to run more than one block in the same instance or keep each block separately running in different instances.

While the project can be extended to multiple clients with minor changes, the current objective is to deal with the job pickup and execution and the performance results. A multi-client DJSF can be analyzed as future work.

DESIGN CONSIDERATIONS

SELECTION OF DHT

The DHT selected is Hazelcast. The main advantage of using Hazelcast is its distributed design. Since the main goal of designing this new scheduler framework is to maintain completely distributed approach without any single master, this is a important point.

Excerpt from [X]:

"Common Features of all Hazelcast Data Structures:

- Data in the cluster is almost evenly distributed (partitioned) across all nodes. So each node carries ~ (1/n * total-data) + backups , n being the number of nodes in the cluster.

- If a member goes down, its backup replica that also holds the same data, will dynamically redistribute the data including the ownership and locks on them to remaining live nodes. As a result, no data will get lost.

- When a new node joins the cluster, new node takes ownership(responsibility) and load of -some- of the entire data in the cluster. Eventually the new node will carry almost (1/n * total-data) + backups and becomes the new partition reducing the load on others.

- There is no single cluster master or something that can cause single point of failure. Every node in the cluster has equal rights and responsibilities. No-one is superior. And no dependency on external 'server' or 'master' kind of concept."

Due to the above properties inbuilt in the hazelcast cluster, the data redundancy and fault tolerance is inbuilt in a hazelcast cluster. There is no single node which can cause a single point of failure.

Implementation Details

The project is implemented in Java for the following reasons:

- platform independence (which allows usage of a heterogeneous distributed system)
- ease of interaction with hazelcast DHT
- support for modular and object oriented and pointer less coding enabling highly maintainable code.

The following components are implemented:

- Client program - To create tasks and populate the job queue
- Executor program - To wait and pick up tasks from the queue and execute them. It also puts back the results into the response queue for the client program to access. Each executor can pickup a maximum of 10 tasks at a time and run them in parallel. This number is configurable as "poolSize" parameter in the DJSF.properties file.
- DistributedQueueHazel program - To run on the hazelcast cluster machines which allows the Client and Executor programs to access the distributed queue on the cluster.
- Task templates - The task template in use is a simple one for the sake of experimentation. However, any Callable, Serializable task can be used with no changes to the model.

A total of ~400 lines of code cover the functionality of all these modules.

Ease of Use

For evaluation purposes, the number of executors and cluster instances and the kind of instances to be used are easily configurable in the EC2 interface provided by Amazon. Further, the DJSF is designed to be able to source the run-time parameters from a DJSF.properties file which is downloaded at startup of the instance. I follow a popular design of hosting the property file on a website (Github) which can be remoted edited and downloaded by the ec2 machines during start up. This way I can easily modulate the number of pollsize,number of tasks,sleep times etc..for the system with ease.

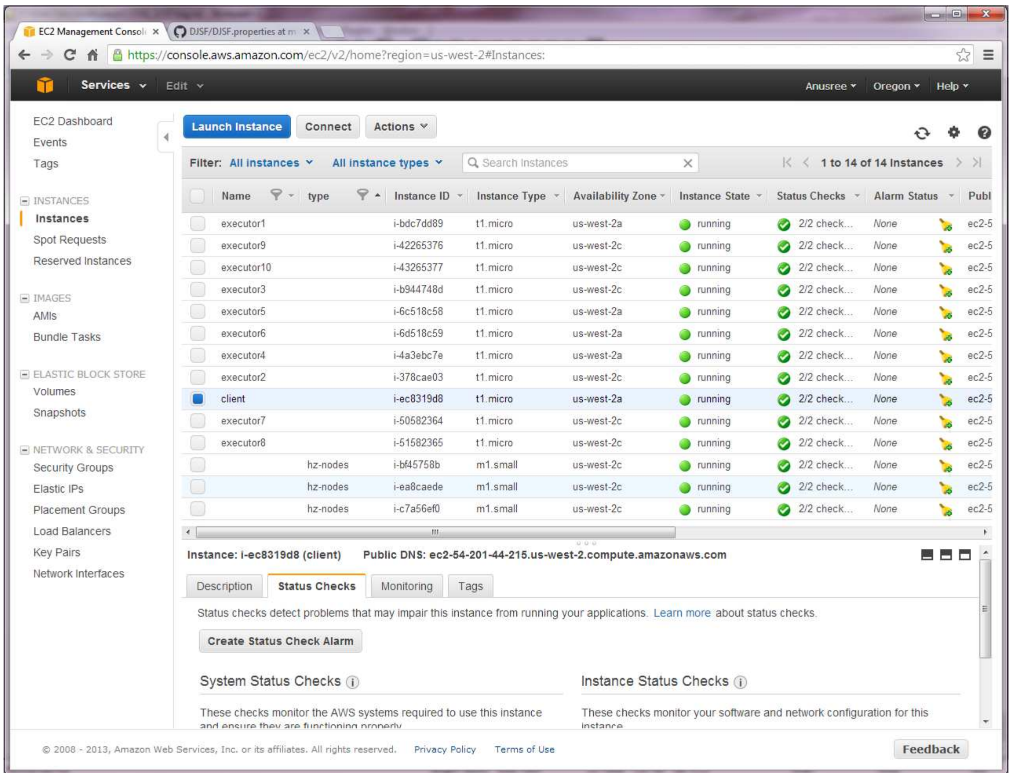
PROJECT RESULTS

Due to cost considerations, low throughput systems were used for the current evaluation of the DJSF task scheduler.

The type of system used for the evaluation results are as follows:

Size	ECUs	vCPUs	Memory (GiB)	Instance Storage (GiB)	EBS-Optimized Available	Network Performance
t1.micro	up to 2	1	0.613	EBS only	-	Very Low
m1.small	1	1	1.7	1 x 160	-	Low
m1.medium	2	1	3.7	1 x 410	-	Moderate
c1.medium	5	2	1.7	1 x 350	-	Moderate

The following screenshot shows the evaluation of the DJSF in a amazon EC2 cluster with 10 executors and 1 client. The hazelcast cluster has a size of 3 instances with the type m1.small. All other instances (executors and client) are t1.micro.



Name	type	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Publ
executor1		i-bdc7dd89	t1.micro	us-west-2a	running	2/2 check...	None	ec2-5
executor9		i-42265376	t1.micro	us-west-2c	running	2/2 check...	None	ec2-5
executor10		i-43265377	t1.micro	us-west-2c	running	2/2 check...	None	ec2-5
executor3		i-b944748d	t1.micro	us-west-2c	running	2/2 check...	None	ec2-5
executor5		i-6c518c58	t1.micro	us-west-2a	running	2/2 check...	None	ec2-5
executor6		i-6d518c59	t1.micro	us-west-2a	running	2/2 check...	None	ec2-5
executor4		i-4a3ebc7e	t1.micro	us-west-2a	running	2/2 check...	None	ec2-5
executor2		i-378cae03	t1.micro	us-west-2c	running	2/2 check...	None	ec2-5
client		i-ec8319d8	t1.micro	us-west-2a	running	2/2 check...	None	ec2-5
executor7		i-50582364	t1.micro	us-west-2c	running	2/2 check...	None	ec2-5
executor8		i-51582365	t1.micro	us-west-2c	running	2/2 check...	None	ec2-5
	hz-nodes	i-bf45758b	m1.small	us-west-2c	running	2/2 check...	None	ec2-5
	hz-nodes	i-ea8caede	m1.small	us-west-2c	running	2/2 check...	None	ec2-5
	hz-nodes	i-c7a56ef0	m1.small	us-west-2c	running	2/2 check...	None	ec2-5

Instance: i-ec8319d8 (client) Public DNS: ec2-54-201-44-215.us-west-2.compute.amazonaws.com

Status checks detect problems that may impair this instance from running your applications. Learn more about status checks.

System Status Checks

These checks monitor the AWS systems required to use this instance and ensure they are functioning properly

Instance Status Checks

These checks monitor your software and network configuration for this instance

© 2008 - 2013, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

For running the experiment with 20 executors, I have used the c1.medium instances for the cluster nodes.

For each instance the Amazon EC2 provides us the public IP address for SSH/SCP, and also an interface via GUI to Start/Reboot/Stop/Terminate each instance. We can save a snapshot(AMI) of the instance using the "Create Image" menu.

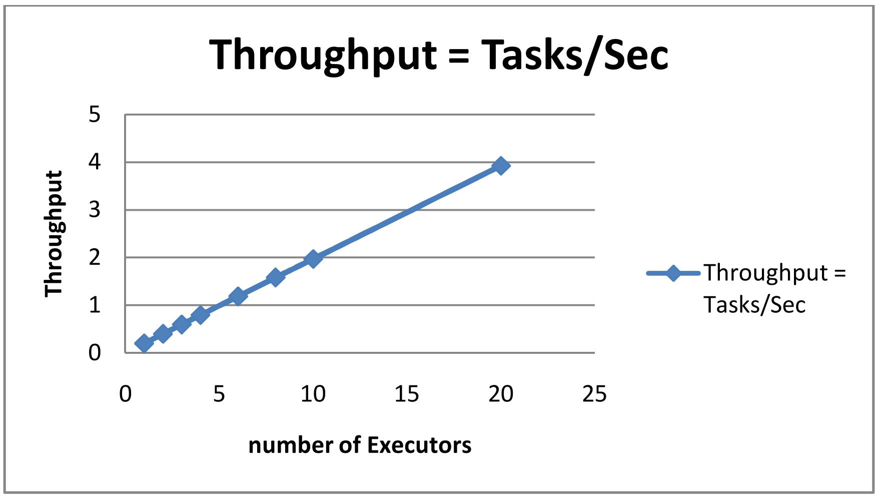
It will be interesting to repeat the experiment for many different types of nodes available in the Amazon EC2 for different types of jobs, but due to time and cost constraints, this level of analysis is reserved for future work.

Performance Evaluation

For evaluating the performance of the task scheduler implemented in this project, The performance tests were run in a EC2 cluster in Amazon. Low performance nodes were used for the hazelcast cluster as well as the executor nodes and client. The following readings are taken using a Amazon EC2 cluster with the minimum configuration with negligible cost. The same tests can be repeated with a high performance cluster to get better results. The tests can also be extended to large number of executors and large number of tasks as future work.

number of Executors	number of Tasks	sleepTime per Task	Time Taken(s) (Tp)	Sequential Time T(s)	Time expected(s) (Tp)	Overhead	Overhead per Task	Overhead Time/Expected Time (%)	Speedup= Ts/(Tp*poolSize)	Throughput = Tasks/Sec
1	100	50	501.1739667	5000	500	1.173967	0.01174	0.234793331	9.976576	0.199532
2	100	50	250.9049637	5000	250	0.904964	0.00905	0.36198546	19.92786	0.398557
3	90	50	150.7047639	4500	150	0.704764	0.007831	0.469842597	29.85971	0.597194
4	120	50	151.2403235	6000	150	1.240324	0.010336	0.826882355	39.67196	0.793439
6	120	50	101.1339876	6000	100	1.133988	0.00945	1.13398763	59.32724	1.186545
8	160	50	101.2306032	8000	100	1.230603	0.007691	1.230603212	79.02749	1.58055
10	200	50	101.5949547	10000	100	1.594955	0.007975	1.594954663	98.43008	1.968602
20	400	50	101.9732236	20000	100	1.973224	0.004933	1.973223595	196.1299	3.922598

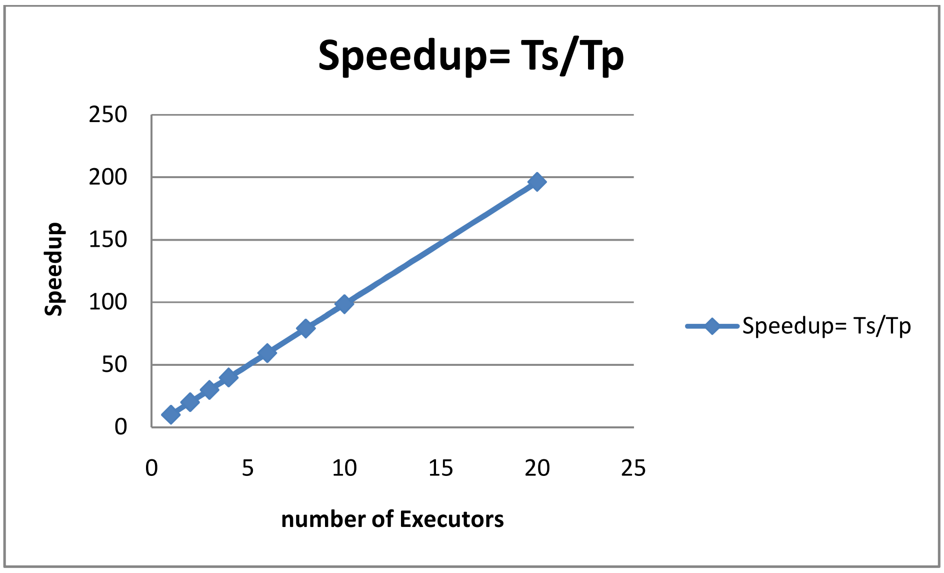
Throughput



The throughput of a distributed system for job scheduling is the number of Tasks that can be completed per second. It is a measure of the performance of the system. For a distributed system, if I scale the size of the system, I can vary the throughput. In DJSF, the number of executors determine the amount of tasks that can be processed per second.

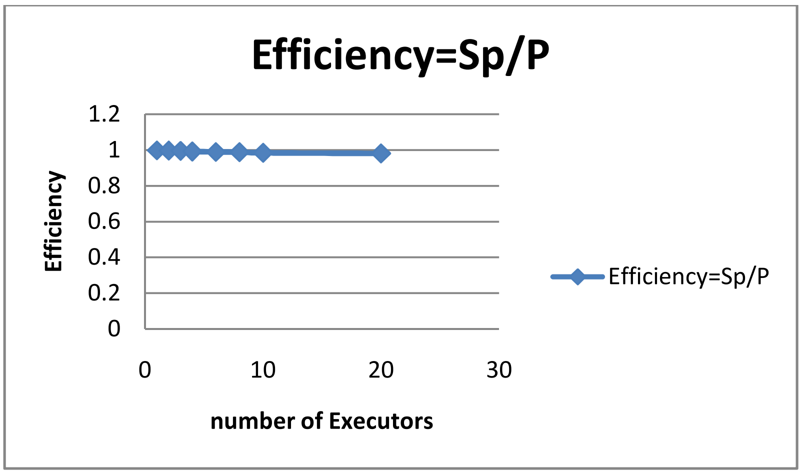
The above graph shows the Throughput measured by varying the number of executors running. The graph shows the ideal linear curve for the actual measurements made. The test was done using a maximum of 20 executors due to cost and time constraints. The reading can be taken for larger numbers of executors to evaluate the maximum throughput of the system. However, it must be noted that according to the memory and network capacity of the instances used for cluster, client and executor, I can get a rise in the throughput for this framework.

SPEEDUP



Speedup is the measure of how much parallelism is incorporated into the system. For a DJSF system, the parallelism depends on the number of executors running. Hence the above graph shows that the speedup achieved scales well with the number of executors running in the system. For higher number of executors, the experiment may at some point encounter a slow down due to network or computing speed bottlenecks. The current evaluation with 20 executors does not show any such performance loss. The speedup measured in near ideal with a minimum speedup of 9.97 for 1 executor and 196.1 for 20 executors each with a pool size of 10.

Efficiency

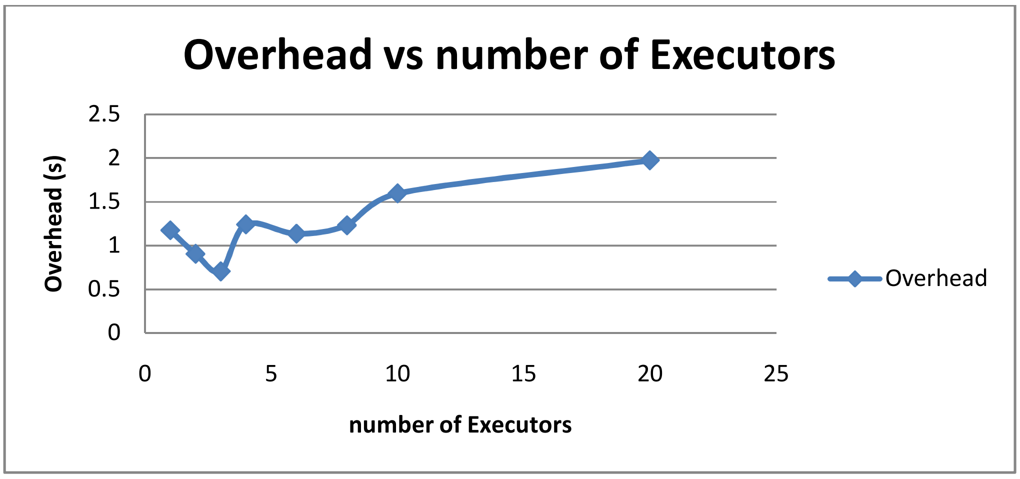


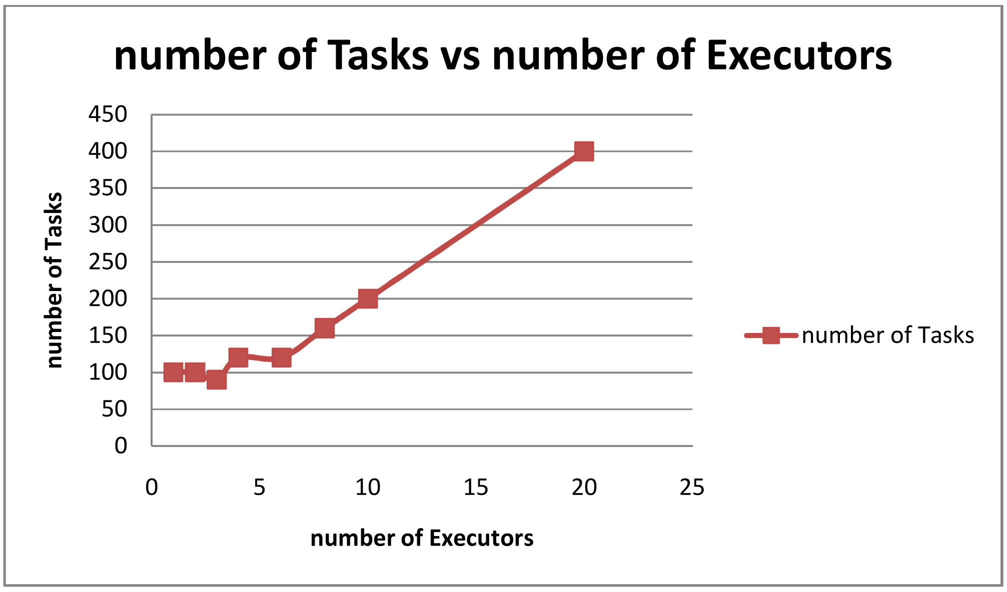
The above graph shows the efficiency measured in the above experiment. The maximum efficiency measured is 99.7% while the minimum efficiency measured is 98.0%. These results are obtained for task with more than 1sec length. For sub-second tasks, I expect that the efficiency will be lower.

Scalability

The overhead measured is the difference between the Tp measured by experiment and the ideal Tp if maximum tasks run in parallel (Note that I limit the maximum tasks run in parallel by the executor to 10. For a homogenous task set (as used in the experiment), I can measure scalability easily. I can easily allow the client to create heterogeneous tasks and put them in the job queue since the Tasks are nothing but java threads implementing the Callable interface. However, in such a experiment, it is not easy to determine the ideal Tp for the experiment.

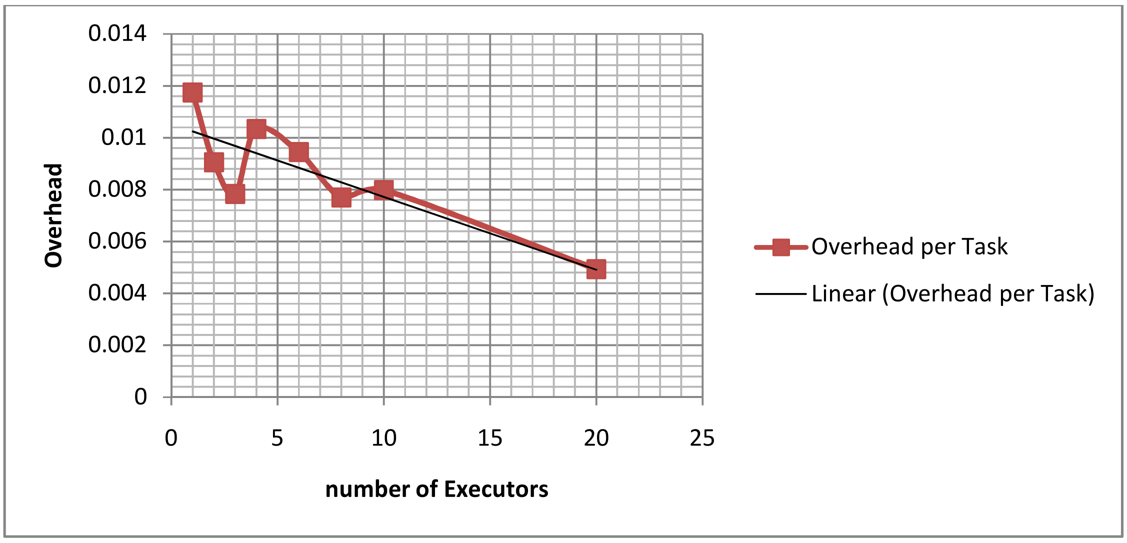
The overhead measure depends on the number of Tasks being queued and the number of Executors being run. Hence, it is seen that the experiment (using variable number of Tasks) displays a decided correlation between the absolute Overhead measured for each experiment vs number of Tasks queued. This can be seen in the following 2 graphs.





Hence I can see that the overhead increases proportionally to the number of Tasks being run. To truly understand how the system scales for higher number of executors, we must remove the dependency on the number of Tasks run. It is not possible to measure the experiment with same number of Tasks since for low number of Executors and high number of Tasks, the experiment becomes too long and for low number Tasks, the experiment becomes irrelevant for high number of Executors.

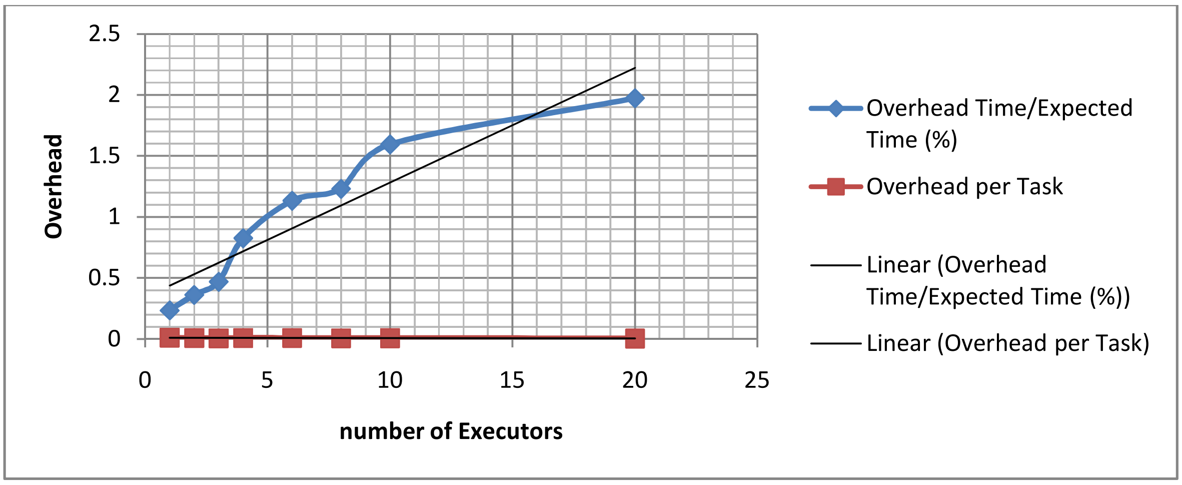
I therefore, take the measure of Overhead time per task and plot that value with respect to the number of Executors. This determines the overhead without the fluctuations due to number of Tasks queued. The following graph shows the relation between overhead per Task vs number of executors. It is seen that as the number of Executors increases in the experiment, the overhead per task has a downward graph. This indicates that the hazel cast cluster with the DJSF task scheduler has good scaling properties. The downward graph is due to the parallelism introduced by increasing number of Executors. The overhead time also gets parallelized and hence over head per task is decreased.



The following graph indicates the Overhead time as a percentage of the ideal Tp for the experiment. i.e

Overhead % =

$$\frac{\mathrm{Tp(actual)} - \mathrm{Tp(ideal)}}{\mathrm{Tp(ideal)}} \mathrm{x}100$$



A scalable system is indicated by such a trend since the overhead time per task decreases as we scale the system up. For all practical purposes the value measured(red line in graph above) is very small and can be considered constant for the system. The Overhead % measured is also shown above (blue line in graph above) is indicative of the trend for the overhead for increasing number of tasks. The trend is linear and not exponential, which is also indicative of a scalable system. However, the experiment must be repeated for larger number of executors to benchmark it versus existing algorithms and this can be done as future work using Amazon EC2 without the current cost-constraint.

Cost

Since this experiment was made at personal cost, I were able to use the free usage tier of Amazon EC2 to get most of the machines required for it. However, since there is a limit of 20 free instances (t1.micro), For the final experiment for 20 executors, some paid nodes were used to run the hazelcast cluster and client. The total cost incurred for the above experiment (including the study phase and configuration phase as well as the bug-fixing phase) was not more than 10USD.

The model is found to be easily scalable to higher number of nodes. Also, the performance results can be further enhanced by using faster and more powerful nodes from the EC2.

Fault Tolerance

To measure the fault tolerance capability of the cluster, the following experiments were made:

numTasks	sleepTime per Task	numExec	Time Taken(s)	Sequential Time T(s)	Time expected(s)	Overhead	Overhead per Task	Overhead Time%	Faulty nodes
200	50	10	100.9577193	10000	100	0.957719	0.004789	0.957719303	0
200	50	10	101.1564134	10000	100	1.156413	0.005782	1.156413381	1
200	50	10	101.7831343	10000	100	1.783134	0.008916	1.783134273	1

200	50	10	101.5949547	10000	100	1.594955	0.007975	1.594954663	1

Since I have configured my hazelcast cluster to have a data redundancy of 1, there should be no data loss for a node failure in 1 node. To simulate the fault, I have rebooted any one hazelcast node during the experiment.

For all cases, I was able to observe that there is no data loss and the experiment was successful with all the tasks running to completion.

However, I can notice that compared to the 1st experiment with 0 faults, the other 3 experiments with different faulty nodes, the overhead time is increased. The data redundancy required the exchange of message between the nodes to redistribute data in the DHT and maintain redundancy. Hence there is some extra overhead introduced due to node failures, though there is no data loss.

Dynamic Resource Scheduling

Extra executor instances can be added on the fly to the DJSF system to increase the speed of the system at high workloads. This is a manual method for the person running the client program to speed up the processing of tasks in the job queue. This method causes an increase in the number of processors utilized for the system and hence provide direct increase in the number of tasks that can be run parallely. The thread poolSize is a parameter in the DJSF.properties file which can also be configured to increase the amount of task threads spawned in each executor process. This may also increase the CPU utilization for I/O bound or frequently blocking task queues.

A completely automatic method of increasing and decreasing the number of executors according to the load of the system is by allowing the hazelcast cluster program "DistributedQueueHazel" to monitor the number of executors versus the job queue size. This can be easily done as future work. In this case, each instance in the cluster can spawn threads for executor and all the instances can run this program. To avoid multiple instances spawning executor threads, a random sleep can be used to check whether the number of executors have already increased and spawn new executor only if it has not. This algorithm will be similar to the random backoff mechanism which is seen in CSMA/CD protocol. In this way I can have a completely distributed dynamic load balancing algorithm with small changes to the DJSF program.

Conclusions

The following conclusions can be drawn from the above study, implementation and analysis of the new Distributed Job Scheduling Framework.

- The above solution is well suitable for a job scheduling framework with low resources at hand. The performance obtained is at very low cost and it has good scaling and fault tolerance capabilities.
- The same system can be used to achieve high performance by allocating high-end machines in cloud; but with a higher cost to the user. This model allows the user to choose between cost and performance and modulate it according to his preferences.
- The solution provided is distributed in all senses and the load balancing can be done either manually by adding or removing executor instances in the EC2 cluster, or dynamically using the random back off mechanism proposed in above section "Dynamic Resource Scheduling".
- I make use of the fault tolerance inherent in the DHT used (Hazelcast) and prove that the failure of nodes do not affect the result though there may be performance implications as the recovery and redistribution of distributed data takes a finite amount of time.

FUTURE WORK

Future Work on this project will involve:

- Performance analysis of the DJSF on a larger scale (more number of nodes, more tasks and more clients) to determine the performance bottlenecks in terms of compute speed or network capacity
- Performance analysis of the DJSF on instances of varied capabilities to resolve any bottlenecks found with the tests on large scales. EC2 provides many different types of instances for my use and it will be interesting to push the algorithm further with better nodes to attain higher performance.
- Dynamic load balancing module: The idea is formed using random backoff algorithm to have dynamic load balancing by adding/removing executor nodes. This can be tested and tweaked to provide optimum load balancing capabilities. The scaling of executor nodes can be done in 2 ways using the amazon EC2 cluster. In one approach, I can scale the system by deploying more instances from the AMI to add executor nodes. In second approach I can scale the system by increasing the amount of executors that can run on each node. The first approach provides horizontal scaling and introduces more processors to the system. The second approach provides vertical scaling by increasing the CPU usage per node - which may be helpful for I/O intensive or resource intensive tasks which have more blocked time.
- Current results are using a homogenous task template with a sleep time of 50s each. However, in real world there will be heterogeneous tasks in the queue with various compute heavy tasks and File I/O heavy tasks of various lengths. Further tests are hence required with such Task queues to understand the capabilities of DJSF to handle such situations.
- The framework must then be used for various real-world applications to determine its performance benchmark against that of other job scheduler algorithms in the market.

REFERENCES

[I] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. MIT Laboratory for Computer Science.

[II] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica. "Looking up data in P2P systems", Communications of the ACM, 46(2):43–48, 2003

[III] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. “Dynamo: Amazon’s Highly Available Key-Value Store.” SIGOPS Operating Systems Review, 2007

[IV] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, Ioan Raicu. ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table in IEEE International Parallel & Distributed Processing Symposium, IEEE IPDPS ’13, 2013

[V] I. Raicu, et. al. “Falkon: A Fast and Light-weight tasK executiON Framework,” IEEE/ACM SC 2007

[VI] J.M. Wozniak, B. Jacobs, R. Latham, S. Lang, S.W. Son, and R. Ross. “C-MPI: A DHT implementation for grid and HPC environments”, Preprint ANL/MCS-P1746-0410, 2010

[VII] Iman Sadooghi, Ioan Raicu. "CloudKon: a Cloud enabled Distributed tasK executiON framework"

[VIII] Fabio V. Hecht, Thomas Bocek, Burkhard Stiller. "B-Tracker: Improving Load Balancing and Efficiency in Distributed P2P Trackers", IEEE P2P 2011 proceedings

[IX] Che-Wei Chang and Hung-Chang Hsiao. "Stochastic Load Rebalancing in Distributed Hash Tables",2011 IEEE 17th International Conference on Parallel and Distributed Systems, 2012 IEEE 18th International Conference on Parallel and Distributed Systems

[X] http://www.hazelcast.com/whatishazelcast.jsp

[XI] Sun Grid Engine Tutorial: http://www.cbi.utsa.edu/book/export/html/29

[XII] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience" Concurrency and Computation: Practice and Experience, Vol. 17, No. 2-4, pages 323-356, February-April, 2005.

[XIII] Swift Workflow System: www.ci.uchicago.edu/swift, 2007.

[XIV] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, M. Wilde. "Swift: Fast, Reliable, Loosely Coupled Parallel Computation", IEEE Workshop on Scientific Workflows 2007.

[XV] I. Foster, J. Voeckler, M. Wilde, Y. Zhao. "Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation", SSDBM 2002

[XVI] J.-P Goux, S. Kulkarni, J.T. Linderoth, and M.E. Yoder, "An Enabling Framework for Master-Worker Applications on the Computational Grid," IEEE International Symposium on High Performance Distributed Computing, 2000.

[XVII] E. Robinson, D.J. DeWitt. "Turning Cluster Management into Data Management: A System Overview", Conference on Innovative Data Systems Research, 2007.

[XVIII] B. Bode, D.M. Halstead, R. Kendall, Z. Lei, W. Hall, D. Jackson. "The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters", Usenix, 4th Annual Linux Showcase & Conference, 2000.

[XIX] S. Zhou. "LSF: Load sharing in large-scale heterogeneous distributed systems," Workshop on Cluster Computing, 1992.

[XX] W. Gentzsch, "Sun Grid Engine: Towards Creating a Compute Power Grid," 1st International Symposium on Cluster Computing and the Grid, 2001.

[XXI] Kay Ousterhout, Patrick Wendell, Matei Zaharia, Ion Stoica, "Sparrow: Distributed, Low Latency Scheduling"