"IUT Job Cracker"
Design and Implementation of a Dynamic Job Scheduler for Distributed Computation

*Fahim Kawsar, **Md. Shahriar Saikat, ***Shariful Hasan Shaikot
Department of Computer Science
*Islamic University of Technology, ***Asian University
**Grameen Software Limited
fahim@dhaka.net, arif9844@yahoo.com , shaikotweb@yahoo.com

ABSTRACT:

This paper presents the design and implementation of a dynamic scheduler for scheduling applications in large-scale, multi-user distributed systems. The approach is primarily targeted at systems that are composed of general-purpose workstation networks. Scheduling decisions are driven by the desire to minimize turnaround time while maintaining fairness among competing applications and minimizing communication overhead. The model handles the task of resource management by dividing the nodes of the system into mutually overlapping subsets. Thus a node gets the system state information by querying only a few nodes. Based on this information the node decides to execute the submitted task locally or remotely conforming optimum resource utilization.

Keywords: Scheduling, distributed systems, Remote Method Invocation,

1. INTRODUCTION:

Every distributed system consists of a number of resources interconnected by a network. Besides providing communication facilities, it facilitates resource sharing by migrating a local process and executing it at a remote node of the network. A process may be migrated because the local node does not have the required resources or the local node has to be shut down. A process may also be executed remotely if the expected turnaround time needs to be better [1]. From a user's point of view the set of available resource in a distributed system acts like a single virtual system. Hence when a user submits a process for execution, it becomes the responsibility of the resource manager of the distributed operating system to control the assignment of resources to processes and to route processes to suitable nodes of the system according to these assignments. A resource can be logical, such as a shared file, or physical such as CPU. While the problem of scheduling parallel applications on distributed computing systems is already well-explored, most existing approaches focus on dedicated, centralized or distributed environments. From a scheduling perspective, first approach addresses a single point of failure whereas the later one increases communication overhead to a great extent [2].

The scheduling model present in this paper typically sits between these two approaches. The entire system is divided into number of subsets equal to the number of nodes and each subset is assigned to every node of the system. A node queries only the nodes of its set to collect most of the nodes state information. When a task is submitted to a node it uses this information to take scheduling decision and then executes the task locally or remotely accordingly. Our model is a semi distributed scheduler that exploits the dynamicity and stability requirements of a good scheduling technique. In addition our method also satisfies the quick decision making capability and provides a balanced system performance with respect to scheduling overhead

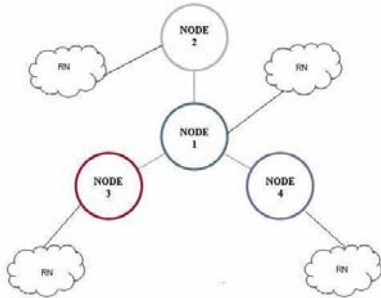
The remainder of the paper is organized as follows. Section 2 outlines the framework on which our artifact is based. Section 3 & 4 discusses about the design and implementation of the scheduler. Section 5 discusses about the performance and future work. Finally, Section 6 concludes the chapter. We name our artifact as IUT JOB CRACKER (IJC). We will use the term IJC as our protocol in rest of the paper.

2. FRAMEWORK MODEL

The underlying framework proposed by Kawsar at el [3] on which our artifact is based can be summarized as:

For N–number, we create N different subsets of size K (K=√N approx) such that each subset overlaps every other subset. Each of the subsets is assigned to different numbers.

[That is, for each number i, we define a subset S_i such that $S_i \cap S_j \neq \emptyset$; For any combination of i and j, $1 \leq i, j \leq N$.]
We divide an N node system into N different mutually overlapping subsets. Each subset is assigned to a different node that is considered as its request set. For selecting a destination node a node query only the member nodes of its request set. If a node query its request set members then that node will be able to get most of the nodes state information because the nodes of its request have the last known state information of their request set nodes which are not included in inquirer request set. Let's see an example.



RN: Rest of the Network (Transparent)
Fig 1: Nodes Connectivity

Considering a system with 13 nodes, we find that the request sets of the node 1, 2, 3 & 4 are

$S_1 = \{1, 2, 3, 4\}$ $S_2 = \{2, 5, 8, 11\}$
$S_3 = \{3, 7, 8, 12\}$ $S_4 = \{4, 7, 9, 11\}$

So the node 1 will exchange its state change messages only with node 2, 3 and 4 and from these nodes it can also acquire the state information of nodes numbered 5, 7, 8, 9, 11, and 12 and can update its system state table. So whenever node 1 is overloaded and a new job is submitted for execution it can migrate its processes to execute remotely to one of the nodes of 2, 3, 4, 5, 7, 8, 9, 11, 12 whose last known state is under loaded [4, 8]. So node 1 does not need to communicate explicitly with all the nodes for making a migrating decision and based on the collected state information it can quickly makes its scheduling decision. The framework reacts towards the addition or removal of a node from the system in the following manner:

When a new node is added to the network, it must—

- Interrogate some active node to get a list of participating nodes.
- Assign itself a unique node number.
- Have its node number placed on all other nodes' list of participating

When a node leaves the network, it must notify all other nodes of its intention. In the meantime, it cannot participate in any communication.

3. DESIGN

In this section we describe the design of our developed artifact for dynamic scheduling. Our design is not limited to Linux, but we will take Linux as an example to explain our design concepts.

3.1 GOALS

Our goal is to design and implement a light-weight general-purpose transparent dynamic scheduling package for existing operating systems. Our design goals include:

- Built with existing operating systems: We want the system to work on general-purpose operating systems. We did not want to write a new operating system, but wanted to leverage investments already made in existing systems. We chose Linux as our platform because it is rapidly growing, totally free, and has source code available.
- Transparent to user applications: We want our package to be general-purpose and completely transparent.
- No kernel modification: By implementing our protocol, we achieved virtually the same level of transparency as kernel patches but avoided changing the kernel itself. This makes it much easier to use.
- Good performance: We do not want to degrade the performance of existing

systems. Especially, we do not want our design to add any run-time overhead to the system.

3.2 ASSUMPTIONS

The basic assumptions of our design are:

1. Assume a homogeneous environment. All machines should have the same architecture, OS installed.
2. Assume the process can continue to access the same files on all machines. The files are on a global file system (such as NFS)
3. Our main concern is dynamic scheduling implementing process migration not crash recovery/rollback.
4. While Linux is not a requirement, our focus is on Linux environments and applications and we will use Linux semantics in this paper.
5. We have used Remote Method Invocation (RMI) approach for servicing the requested job. So only java object can be serviced.

3.3 BASIC APPROACH:

IJC does dynamic scheduling by the basic idea given in the section 2. For remote execution of the processes we have used REMOTE METHOD INVOCATION (RMI). The overall approach can be summarized as:

- Install and load “Request Set Generator” on all machines that generate the request set of the respective machines.
- Implement the global file system (NFS). This can be done during booting time or later on explicitly when IJC is called.
- Install and load” Load Calculators” on all machines. These modules run in kernel space but do not require kernel modification. These modules calculate the system load require for decision making.
- Hosts communicate via virtual network address translation.
- Each of the machines is configured as both request client and request server. Each of the machines starts listening to a particular port for request.
- Request a process or a group of processes to be executed by invoking the graphical interface of IJC.
- IJC decides whether to execute the requested process remotely or locally by viewing the system environment status.
- IJC then perform RMI to execute the process employing stub processes.
- After execution of the requested job at the remote or local node the results are sent back to the home node and displayed graphically to the user.

3.4 USER INTERFACE

IJC: the user should be able to specify which process(s) to execute and the necessary parameter to the respective processes. Figure 2 shows the graphical interface of the IJC.

Job Report: The user should be displayed the result of the requested task. IJC does so by displaying the job report of each of the task in a separate dialog box. The template of Job Report dialog box is shown in Figure 3.

Figure 2: Graphical Interface of IJC

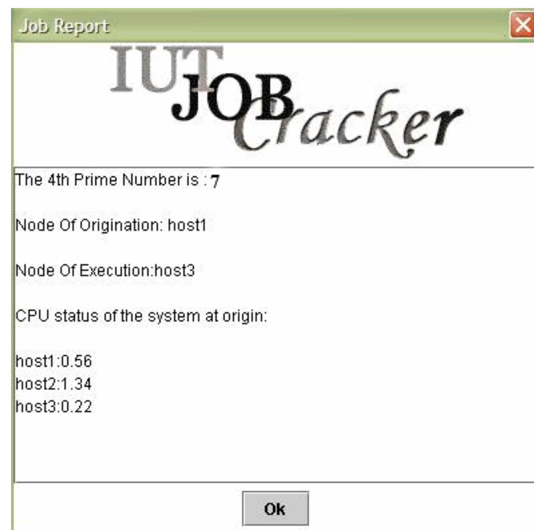


Figure 3: Job Report Dialog Box of IJC

4. IMPLEMENTATION

In this section we describe our implementation of IJC as a user program on Linux 2.4.x/i386. The basic picture of the entire process is, needless to say, composed of two phases: selection of node and execution of the process at selected node. We first give an overview of the implementation, and then describe the key components.

Basically to reveal the entire implementation phase we need to begin from the underlying algorithmic approach towards distributed scheduling. While installing a new machine to the system we provide a module that creates the request set of that machine with respect to the entire system and updates the necessary file. Each of the machines is then automatically configured to start a script and to implement NFS while rebooting. This script performs the recording of system load periodically and invokes modules that gather the system wide load picture which facilitates appropriate decision making for node selection to execute a process. The main components of the scheduler are:

1. REQUEST SET GENERATOR:

Source File: Set.cpp.
Binary File: makeset.
Consequence: Creates a file named homeset.dat in /home directory and updates /etc/exports and /etc/fsatb

This is the module that actually performs the request set creation for each of the node. This module implements the algorithm presented in section 2 .This binary module causes a new file creation in the /home directory names homeset.dat which contains the symbolic names of the hosts in the request set. An example of homeset.dat file for 3 node system is :

host1/home/homeset.dat
host1
host2

host2/home/homeset.dat
host2
host3

host3/home/homeset.dat
host1
host3

This module also updates the /etc/exports file which causes the /home directory of the respective node to be exported to the nodes of the homeset.dat file. An example of the /etc/exports file of nodes may be for 3 node system:

host1/etc/exports
/home host2(rw)

host2/etc/exports
/home host3(rw)

host3/etc/exports
/home host1(rw)

This module also updates the /etc/fstab file which causes the /home directory of the respective nodes of the homeset.dat file to be mounted in the respective node at booup. An example of the /etc/fstab file entry for this of host1 node may be for 3 node system:

host1/etc/fstab entry
host2:/home /mnt/host2 nfs defaults 0 0

host2/etc/fstab entry
host3:/home /mnt/host3 nfs defaults0 0

host3/etc/fstab entry
host1:/home /mnt/host1 nfs defaults 0 0

2. LOAD CALCULATOR:

Source File: viewusage.cpp, calrs.cpp, calglobal.cpp
Binary File: viewusage, calrs, calglobal
Script File: usage
Consequence: Two file localenvinfo.dat and globalenvinfo.dat are created at the /home directory of the respective nodes. Two temporary file lookup and cpu_use are created at /home and /bin directory respectively.

The script file name usage is run in background whenever the system is up. This scripts periodically measures the cpu use (load) of the node and writes it in raw format at cpu_use file. The viewusage module parse this file accordingly and extracts the current load and writes this information at /home/lookup file. The script then calls calrs module which reads the local lookup file and the remote lookup files of the nodes that are in it’s request set and write each of the nodes load along with their name in

/home/localenvinfo.dat file. The usage script then causes the calglobal module to be called. This module reads the local lookup file and the remote localenvinfo.dat file of all nodes that are in it's request set and writes all these information at globalenvinfo.dat file in it's /home directory. Thus each of the nodes gathers the picture of the global system load.

Example of cpu_use file is:

6:04pm up 21 min, 2 users, load average: 0.79, 0.27, 0.21

Example of lookup file is:
0.79

Example of localenvinfo.dat file is
host1 0.87
host2 0.67

Example of globalenvinfo.dat file is
host1 0.87
host2 0.67
host3 0.45

3. The IJC Interface:

At this point each of the machine has the system wide load information. Now when IJC is invoked a script file is run at background which causes all the java file to compiled and makes the client and server process ready for listening to particular port for request. Typically we register the RMI service in a particular port and run all the server stubs at the background.

Now when the job is submitted through the graphical interface of IJC, IJC internally reads the globalenvinfo.dat file and decides whether to execute the request locally or remotely. We may use a threshold value here to select the best node for processing. If all the machines in the system are lightly loaded including the home node then it is better to execute the process locally to avoid communication overhead. If it decides to execute the process remotely then it causes the server stub of that node to be called and execute thereby. Java RMI handles the entire process migration mechanism. For each of the request the IJC creates a user level thread thus simultaneously several task can be given to the node. After execution of the task either remotely or locally the result is transmitted to the IJC interface which is displayed through job report dialog box.

5. PERFORMANCE ANALYSIS AND FUTURE WORK

The performance of the model is significant from overhead perspective. For a traditional N node distributed system each node need to exchange (N-1) messages (Information Exchange Policy) [4] for updating system management table thus making a scheduling decision. The mot attractive feature of this model is that we need to exchange only (K-1) messages where $K=\sqrt{N}$ (approx) to gather system wide information. This reduces network traffic to a great extent. It also provides greater reliability then centralized approach or distributed approach. As node crash does not result in the down of the entire system. Rather the node misses only partial information for making scheduling decision. The model provides the scheduling decision influencing information quite fairly. We have tested our model with two very popular and frequently used tasks namely prime number generation and Fibonacci number generation. We have found that the scheduler always selects the best node (with lowest load) for execution of the processes and displays the result with minimum turn around time.

Currently our model is capable of handling only specific java objects as because of its inherent dependency on RMI. However the idea can be extended to support any kind of objects. For example we may use CORBA (Common Object Request Broker Architecture) to facilitate support for any kind of processes. We are currently working on incorporating CORBA in our architecture. Hopefully we will be able to facilitate services for all processes in near future.

6. CONCLUSION:

The paper discussed about the design of a dynamic scheduler that reduces the communication overhead involved in decision making for resource management. It is visible that from number of message exchanging perspective our designed model provides a significant performance. As the performance of a distributed system is heavily influenced by the scheduling mechanism, the model presented in this paper may lead to a balanced system performance as scheduling decision is performed quite fairly with less communication overhead.

REFERENCES:

[1] J. XU AND K. HWANG. Dynamic load balancing for parallel program execution on a

message passing multi computer. Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing pp. 402–406, 1990.

[2] ANDREW S. TANENBAUM, "Distributed Operating Systems", Prentice Hall, 1995

[3] Fahim Kawsar, Shariful Hasan, Shahriar Saikat, "An Efficient Dynamic Scheduling Algorithm in Distributed System", Proceedings of ICCIT-2002

[4] S. ZHOU. A trace driven simulation study of dynamic load balancing. IEEE Trans. Software Engineering 14(9): 1327–1341, Sep. 1988.

[5] J. M. SMITH. A survey of process migration mechanisms. *ACM Operating Systems Review* 22(3):28–40, Jul. 1988.

[6] SILBERSCHATZ, A. and PETERSON, J.L., "Operating System Concepts," Addison-Wesley, Alternate edition, 1988.

[7] MAEKAWA, M., "A sqrt(n) algorithm for mutual exclusion in decentralized systems," ACM Transactions on Computer Systems, vol 3, no. 2, May 1985, pp. 145-159

[8] PRADEEP K SINHA", Distributed Operating Systems, Concepts and Design", IEEE Press, 1998