1993

A system model for distributed job scheduling : the distributed job management system

Michael Kenneth Nemeth
Lehigh University

AUTHOR:

Nemeth, Michael Kenneth

TITLE:

A System Model for Distributed Job Scheduling: The Distributed Job Management System

DATE: January 16, 1994

A System Model for Distributed Job Scheduling:

The Distributed Job Management System

by

Michael Kenneth Nemeth

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Department of Electrical Engineering and Computer Science

Lehigh University

Bethlehem, Pennsylvania 18015

December 10, 1993

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

12/7/93

Date

Thesis Advisor

Chairperson of Department

Table of Contents

List of Tables v

List of Figures v

Abstract 1

1. Introduction 2
    1.1. Organization 3
    1.2. Distributed Job Scheduling 4
    1.3. The Distributed Job Management System (DJMS) 5
    1.4. Future Research 5

2. Literature Survey of Distributed Job Scheduling 7
    2.1. Components of Distributed Job Scheduling 7
    2.2. Sender Initiated Algorithms 10
    2.3. Receiver Initiated Algorithms 12
    2.4. Symmetrically Initiated Algorithms 13
    2.5. Adaptive Algorithms 14
    2.6. A Comparison of Algorithms 16

3. The Distributed Job Management System (DJMS) 19
    3.1. DJMS Motive and Vision 19
    3.2. DJMS Hardware and Software Model 21
    3.3. DJMS Concepts and Terminology 23
    3.4. DJMS Overview 25
    3.5. DJMS Operating Components 27
        3.5.1. Distributed Job Request Facility (djrf) 28
        3.5.2. Distributed Job Status Facility (djsf) 29
        3.5.3. Distributed Job Cancel Facility (djcf) 30
        3.5.4. Distributed Job Configuration Change Facility (djccf) 32
        3.5.5. Distributed Job Monitor Facility (djmonf) 33
        3.5.6. Distributed Job Management Facility (djmanf) 35
    3.6. A Summary of the DJMS Components 38
    3.7. DJMS Fault Tolerance 39
    3.8. DJMS Throughput Analysis 42

4. Conclusion and Future Work 47

List of References 49

Appendix 51
Distributed Job Request Facility (djrf) Manual Page 51
Distributed Job Status Facility (djsf) Manual Page 56
Distributed Job Cancel Facility (djcf) Manual Page 59
Distributed Job Configuration Change Facility (djccf) Manual Page 62
Distributed Job Monitor Facility (djmonf) Manual Page 65
Distributed Job Management Facility (djmanf) Manual Page 67
Distributed Job Start (djstart) Manual Page 70
Distributed Job Stop (djstop) Manual Page 71

Vita 72

List of Tables

Table 1 - Algorithm Policy Comparisons 17

Table 2 - DJMS Concepts and Terminology 23

Table 3 - A Summary of the DJMS Components 39

Table 4 - A Summary of the DJMS Throughput Analysis 44

List of Figures

Figure 1 - Unbalanced Distributed System 3

Figure 2 - Sender Initiated Algorithms 11

Figure 3 - Receiver Initiated Algorithms 13

Figure 4 - Symmetrically Initiated Algorithms 14

Figure 5 - Adaptive Algorithms 16

Figure 6 - Job Submission into DJMS 29

Figure 7 - Job Status and Tracking in DJMS 30

Figure 8 - Job Cancellation in DJMS 31

Figure 9 - Configuration Change in DJMS 33

Figure 10 - Load Sharing in DJMS 35

Figure 11 - Stability Control in DJMS 38

Figure 12 - Fault Tolerance in DJMS 41

Figure 13 - DJMS Throughput Analysis 45

Figure 14 - DJMS Response-Time Analysis 46

Abstract

Advances in high speed local area networks and powerful workstations have spurred the movement of both business and academic computing from a traditional centralized mainframe environment to a decentralized distributed processing environment. By sharing system resources on loosely coupled processors, the computing cost per user can be significantly decreased. In order to achieve and support a high-level of resource sharing and provide adequate user response-time, efficient techniques in distributed job scheduling become a requirement. A large portion of the distributed job scheduling problem resides in the efficiencies of: (1) job or task distribution amongst processors and resources and (2) the method and frequency of communications. This thesis reviews distributed job scheduling and emphasizes its basic components and policies. After discussing an overview of distributed job scheduling techniques as shown in recent literature, a system model for distributed job scheduling, The Distributed Job Management System (DJMS) is presented. The DJMS presentation provides comprehensive information on system components, operations, fault tolerance, and throughput analysis.

"By wisdom a house is built, and through understanding it is established; through knowledge its rooms are filled with rare and beautiful treasures."
- Proverbs 24:3,4

Chapter 1. Introduction

With the availability of high speed local area networks and powerful desk top workstations, both business and academic computing have begun to migrate from the traditional and centralized mainframe to a decentralized and distributed workstation processing environment. The benefits of such a distributed architecture are vast. With the sharing of resources such as file systems, printers, and processors, the computing cost per user has significantly decreased. On the contrary, as user and business applications grow they tend to require more processing power than one workstation or processor can provide. In order to meet this processing demand, distribution of job or task loads, through methods of sharing and balancing among networked processors, becomes inevitable. As figure 1 shows, some processors can become overloaded while others remain slightly or moderately loaded and under-utilized. Without proper load distribution, support of application growth while providing adequate performance cannot be achieved.

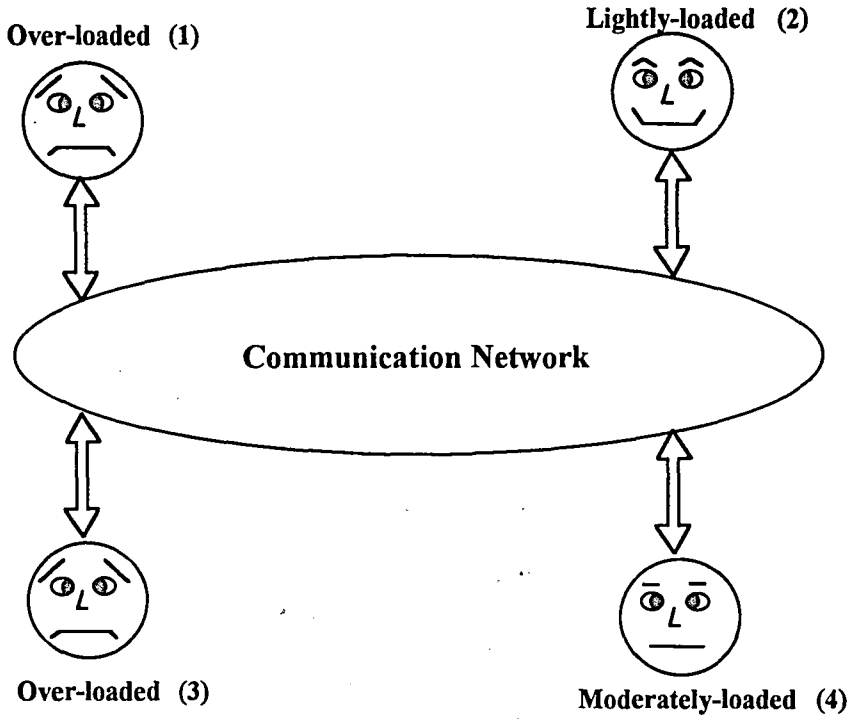

Figure 1 - Unbalanced Distributed System

1.1. Organization

A substantial area of research focuses on the techniques and problems in distributed job scheduling algorithms[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]. A goal of this research is to understand the efficiencies of job or task distribution among processors and their effects on system resources and the underlying network. Typically, an analysis of processor and communication overhead as well as cost are studied. This thesis will focus on currently used distributed job scheduling algorithms and will present a distributed job scheduling model, The Distributed Job Management System (DJMS). The DJMS presentation provides comprehensive information on system components, operations, fault tolerance, and throughput analysis.

1.2. Distributed Job Scheduling

Distributed job scheduling can simply be described as an algorithm where jobs or tasks are transparently migrated and subsequently executed at neighboring processors with best intentions to (1) maximize global system throughput and (2) meet job or task response-time requirements[1,3]. In distributed job scheduling, the basic premise is to transfer jobs or tasks from heavily loaded processors, where service response-time is poor, to slightly or moderately loaded processors, therefore improving response-time. By doing so, system resources on under-loaded processors that would have been otherwise wasted are more efficiently utilized to an advantage. As the designer implements such an algorithm, many system issues and concerns must be considered. For instance, which processors are slightly, moderately, and heavily loaded? Which processors have access to the resources needed to complete given jobs? Which processor can service a job in order to fulfill the desired response-time? Which processors are up and available? As one might conjecture, the construction of a distributed job scheduling algorithm is extremely elaborate and requires much research and development in order to perfect it.

In chapter two of this thesis, recent technical literature covering today's technology of distributed job scheduling is presented. This chapter discusses the structures, components, and policies of distributed job scheduling and concentrates on the explanation and study of the most predominate dynamic and adaptive distributed algorithms (i.e. the sender initiated, receiver initiated, symmetrically initiated, and adaptive algorithms)[1,3,4,5,7,9,10,14]. At closing of chapter two, a comparison between algorithms is presented.

1.3. The Distributed Job Management System (DJMS)

The Distributed Job Management System (DJMS) is a centralized and adaptive distributed job scheduling system that performs both load sharing and load balancing activities for background processing on locally networked *UNIX* processors. In chapter three of this thesis, the Distributed Job Management System is the main focus. As an introduction, motives and visions for developing DJMS are discussed as well as details about its development and execution platforms. At the heart of the third chapter, the DJMS operating components are presented. DJMS is composed of six components or commands, three of which are the user level interface, while the remaining are strictly for administration and system operation. The Distributed Job Request Facility (djrf) supports user submission of background jobs for execution. The Distributed Job Cancel Facility (djcf) supports user cancellation and the Distributed Job Status Facility (djsf) supports status for background jobs that have been submitted. The Distributed Job Configuration Change Facility (djccf) supports an administration feature where a load sharing configuration can be dynamically modified. The Distributed Job Management Facility (djmanf) and Distributed Job Monitor Facility (djmonf) are daemon processes which control the overall distributed job scheduling. As each of the components are described, comprehensive information regarding their purpose, usage, and interaction is presented. At closing of chapter three, DJMS's fault tolerance is discussed and a case analysis on system throughput is presented.

1.4. Future Research

As more real-time distributed systems come about, a more sophisticated breed of distributed job scheduling algorithms must be devised and perfected. Typically in real-time systems, jobs or tasks are restricted to extremely tight completion deadlines, which cannot be missed. In order to accommodate such a requirement in a distributed system,

more global-state information must be gathered and maintained to (1) better depict the load at each processor and (2) accurately derive task completion[2,6,8,11,12,13,15]. Algorithm designers are beginning to focus on such a solution. In closing, chapter four presents some concluding thoughts about distributed job scheduling and the Distributed Job Management System (DJMS).

Chapter 2. Literature Survey of Distributed Job Scheduling

An excellent place to begin reviewing distributed job scheduling is in the recent technical literature. The purpose of such a survey is two-fold. It provides the reader with the necessary background and understanding of distributed job scheduling while uncovering some of the writer's influences and motives for this thesis. This survey investigates literature dated 1988 to the present, and while not inclusive, covers the most prevalent dynamic and adaptive distributed job scheduling algorithms. For each dynamic and adaptive algorithm presented, the following components are described: transfer policy, selection policy, location policy, information gathering policy, stability control policy, and local scheduling policy. At the conclusion of this chapter, some pedagogical comparisons among algorithms will be presented.

2.1. Components of Distributed Job Scheduling

Most distributed scheduling algorithms today can be classified as either static, dynamic, or adaptive[1,2,3,10,11,13]. A static distributed job scheduling algorithm typically distributes jobs or tasks based on decisions made by the local processor and only the local processor with no global system-state information gathering. The assignment of jobs or tasks to processors is managed in a low overhead, random manner. Much research has demonstrated that this class of algorithms consistently results in poor scheduling decisions under heavy system loads. The dynamic class of algorithms distribute their jobs or tasks based on decisions derived globally by gathering system-state information. Tasks are assigned to processors by use of a high overhead decision making process where peer processors work cooperatively. Much research has demonstrated that this class of algorithms out-performs its static counterpart. The adaptive class of algorithms are for the most part a specialized set of dynamic algorithms. This class of algorithms typically

modify their activities to improve their scheduling decisions, therefore globally maximizing the system throughput.

Distributed job scheduling algorithms are either centralized, decentralized, or hierarchical in structure[3,11,13]. For the most part, centralized algorithms dictate the placement of jobs or tasks from slave processors through a centralized master processor. The master processor has global knowledge as to where job or task distribution should occur on the system. Centralized algorithms are less reliable due to one point of system failure, and also tend to create a bottleneck in the job or task distributing activities. For the most part, hierarchical algorithms funnel the jobs or tasks from child processors through a parent processor. The parent processor has global knowledge as to where job or task distribution should occur on the system. Hierarchical algorithms are more reliable, for if one part of the processor tree fails, job or task distributing activities can coexist on other processor sub-trees. The hierarchy tends to eliminate the bottleneck in the job or task distributing activities as well. Decentralized algorithms migrate jobs or tasks between peer processors offering the most effective solution to job or task distribution. This effectiveness however, tends to increase overhead and complexity.

The distribution of jobs or tasks can be either be preemptive or non-preemptive[2,3]. Preemptive task transfer can be extremely expensive due to the collection of an executing task's state. Freezing of a task process state requires gathering of memory, open files, messages, timers, etc.. Typically, the expense of such a complex task outweighs any benefit. Further research needs to be performed in this area. On the contrary, non-preemptive task transfer is rather simple. Jobs or tasks are transferred between processors, without current state information.

In all distributed scheduling algorithms, a key component is a load index[2,3,9]. A load index is typically information that indicates the magnitude of work at some processor

and correlates well with task response-times. Many load indices are used in distributed task scheduling. Load indices such as length of CPU queue, memory consumption, process context-switch rate have been studied. The results of these studies have demonstrated differences in their effectiveness.

In the current technical literature, most distributed job scheduling algorithms are described and explained through six basic components: the transfer policy, selection policy, location policy, information gathering policy, stability control policy, and local scheduling policy[1,2,3]. The first component, a transfer policy, decides when an attempt should be made to migrate a job or task from one processor to another. The transfer policy typically identifies which processor will act as the sender of the job or task and which processor will act as the receiver. Most often the load index at each processor is compared to a load threshold to determine the processor's role. The second component, a selection policy, decides which job or task to actually transfer. Most often a newly arriving job or task will be selected for transfer. Sometimes a job or task that transfers will either be a task that does not require local resources, will execute for a long period of time, and will encounter the least amount of overhead. The third component, a location policy decides to which processor a job or task, which is eligible for migration, will be transferred. This policy will attempt to find the optimum destination, i.e., the processor that has possibly the shortest CPU queue length. The fourth component, an information gathering policy decides what information the location policy needs in order to make its decisions and how this information is to obtained. Typically, information is gathered on demand, periodically, or upon a change in the system's state. The goal is to obtain sufficient information at the minimum cost in terms of communications and processing. Naturally, if the overhead incurred by gathering the information becomes large, it may offset part or all of the gains obtained by transferring a job or task. The fifth component, a stability control policy ensures that a task is not migrated endlessly. It guarantees that a

job or task will eventually be serviced and executed by a processor. The sixth and last component, a local scheduling policy indicates the order in which tasks waiting at a processor will be serviced and executed.

2.2. Sender Initiated Algorithms

In sender initiated algorithms, job scheduling is initiated by an overloaded processor (sender) trying to send a job to an under-loaded processor (receiver)[1,2,3,6,11]. As for the algorithm's transfer policy, when a new job is queued on the sender processor, a load index, such as a CPU queue length is checked to determine whether the new job overflows the processor's load threshold. Typically, a processor is identified as a receiver when accepting the new job does not cause the processor's load index to exceed the load threshold. As for the algorithm's selection policy, only newly arriving jobs are considered for transfer or migration. As for the algorithm's location policy, three common techniques have been utilized. First, a job or task is transferred to a receiver processor in a random manner. In this scheme, no global-state information is gathered and used in the transfer decision making. Second, a transfer of job or task to a receiver processor with the shortest CPU queue length has been used. In this scheme, global-state information is gathered and used in the transfer decision making. Third and last, a transfer of job or task to a receiver processor with a CPU queue length less than some threshold limit has been used. This scheme also utilizes global-state information while making transfer decisions. As for the algorithm's information gathering policy, a demand-driven approach is utilized. Information is typically gathered as a processor becomes a sender. Once a sender, information is demanded from other receiver processors to make a decision as to which location will receive the transfer. Generally, sender initiated transfers only migrate jobs or tasks in a non-preemptive fashion. This is due to the fact that the initiation of a sender transfer is always triggered by a new arrival of

a task. As for the algorithm's stability control, not much is enforced. As processors become increasingly busy, CPU time and resources are under-utilized due to endless transfer of jobs or tasks from one processor to another. This algorithm does not adapt well to the heavily loaded situation. As for the algorithm's local scheduling policy, a first-in-first-out queuing scheme is usually followed. As figure 2 shows, during task arrival, processor 1, an overloaded (sender), attempts to locate an under-loaded (receiver). Initially, processor 3 was polled and determined to be an inadequate receiver. Lastly processor 2, an under-loaded (receiver), was found and accepted the task transfer.

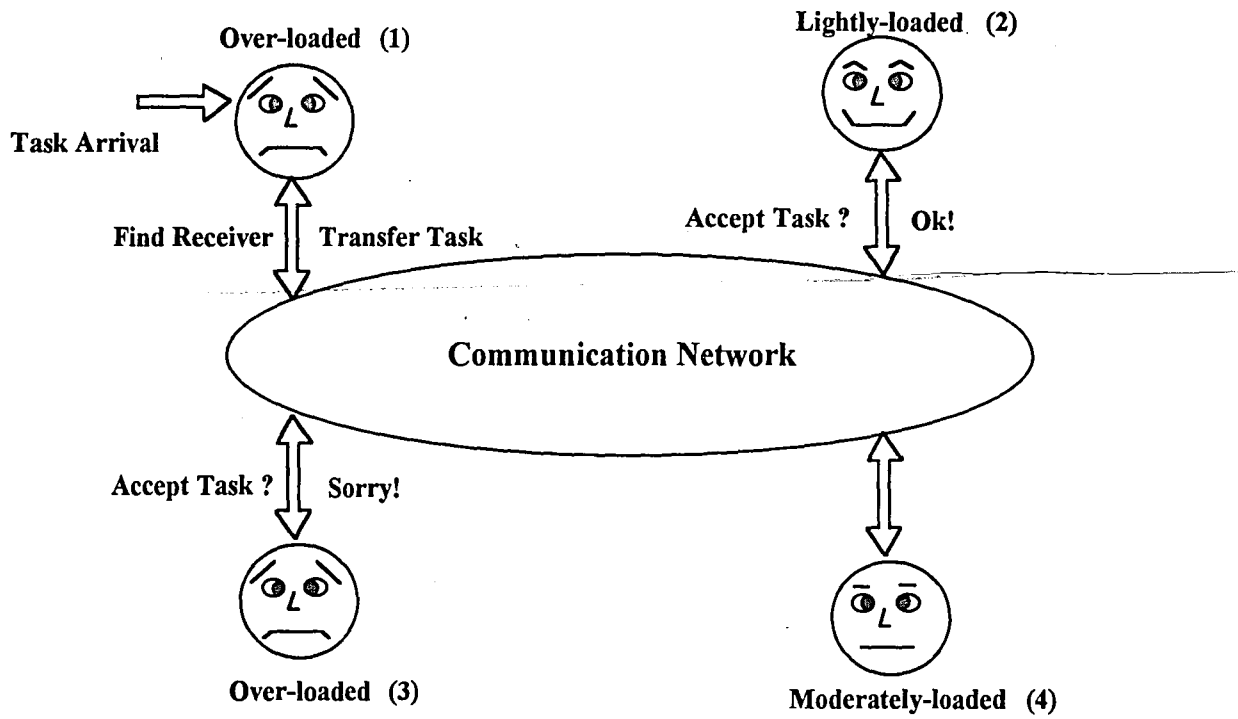

Figure 2 - Sender Initiated Algorithms

2.3. Receiver Initiated Algorithms

In receiver initiated algorithms, distributed job scheduling is instituted when a receiver processor becomes lightly loaded and requests a job or task from a overloaded (sender) processor[2,3,8]. As for the algorithms transfer policy, a processor's CPU queue length is monitored. As the CPU queue length drops below its threshold the processor becomes a receiver and is eligible to accept jobs or tasks. Typically, a processor is identified as a sender when the processor's load index exceeds the load threshold. As for the algorithms selection policy, all jobs or tasks are considered for transfer or migration. As for the algorithm's location policy, one common technique has been utilized. A random poll for a job or task to a sender processor has been used. In this scheme, no global-state information is gathered and used in the transfer decision making. Typically, the sender processor will transfer a job or task as long as its CPU queue length does not fall below its load threshold. As for the algorithm's information gathering policy, a demand-driven approach is utilized. Information is typically gathered as a processor becomes a receiver. Once a task departs, information is demanded from other sender processors to make a decision as to which sender location will be chosen to execute the transfer. Commonly, receiver initiated transfers can migrate jobs or tasks in preemptive fashion. This is due to the fact that the request for a sender transfer is always triggered by a departure of a task at the receiver processor. Therefore, tasks already executing at an overloaded sender maybe chosen to migrate. As for the algorithm's stability control, not much is enforced, but it is not likely for instability to occur. As receiver processors become increasingly busy, less CPU time and resources are wasted on polling for overloaded sender processors. The receiver processor already has an adequate load and does not require added or additional work. In fact, if a receiver processor becomes too overloaded, it may change roles and become a sender. This algorithm adapts rather nicely by default to the heavy loaded situation. As for the algorithm's local scheduling policy, a first-in-first-out queuing

scheme is usually followed. As figure 3 shows, during task departure, processor 2, a lightly loaded (receiver), attempts to locate an overloaded (sender). Initially, processor 4 was polled and determined to be an inadequate sender. Lastly processor 1, an overloaded (sender), was found and subsequently transferred a desired task.

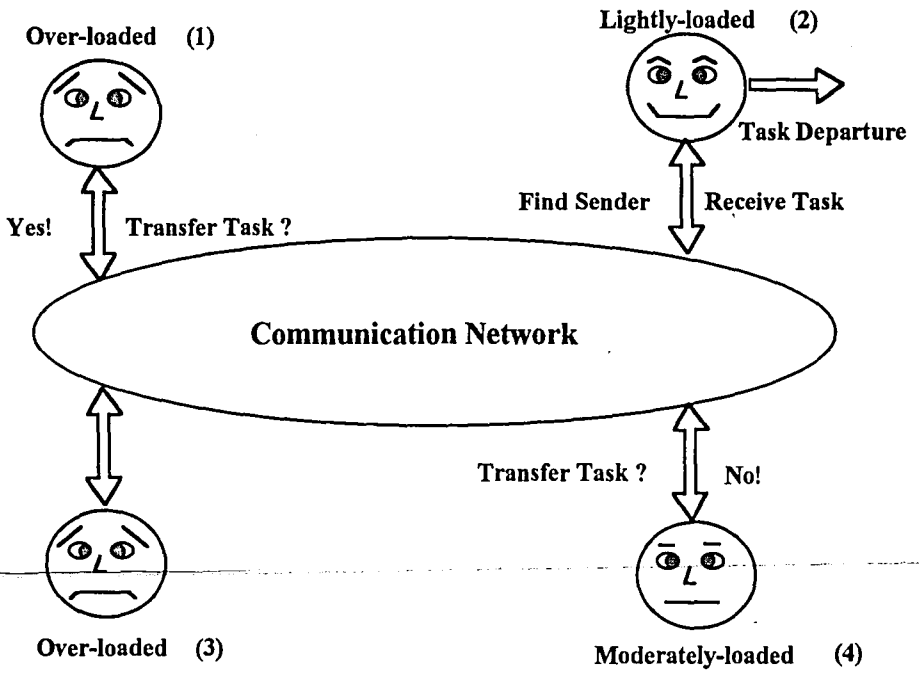

Figure 3 - Receiver Initiated Algorithms

2.4. Symmetrically Initiated Algorithms

In symmetrically initiated algorithms, distributed job scheduling is instituted when a receiver processor becomes slightly loaded or when a sender processor becomes heavily loaded[3,7,12]. Typically, during lighter system loads the sender initiated portion of the algorithm is more efficient. During heavier system loads, the receiver initiated portion of the algorithm is more efficient. For the most part, this class of algorithms exhibits the same strengths and weaknesses of its sender and receiver initiated counterparts. As for the algorithms six policies, it is merely a combination of the policies previously described in the sender and receiver initiated algorithms. As figure 4 shows, during task arrival,

processor 3, an overloaded (sender), attempts to locate a slightly or moderately loaded (receiver). Processor 4, a moderately loaded (receiver), was found and accepted the task transfer. Meanwhile, during task departure, processor 2, a lightly loaded (receiver), attempts to locate an overloaded (sender). Processor 1, an overloaded (sender), was found and subsequently transferred a desired task.

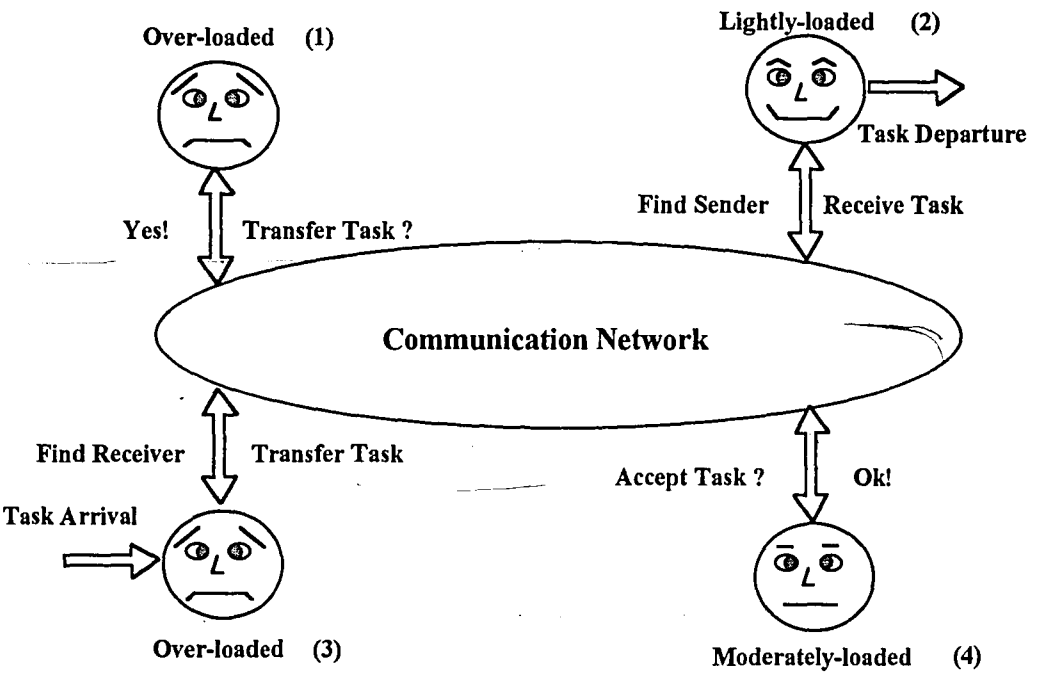

Figure 4 - Symmetrically Initiated Algorithms

2.5. Adaptive Algorithms

In adaptive algorithms, distributed job scheduling is performed by either the overloaded sender or slightly loaded receiver[3,12,14]. For the most part, this class of algorithms is comparable to symmetrically initiated algorithms with a few minor improvements in both communication overhead and stability control. Typically, more sophisticated system-state information about the participating processors in the distributed system is gathered and maintained. As load distributing events occur, each processor categorizes its peers into the following three distinct groups: sender/overloaded,

receiver/under-loaded, and sender or receiver/ok-loaded. Typically, these groups are maintained as lists on each processor and are updated as processors interact through messages. As for the algorithms transfer policy, each processor monitors two local threshold values which are usually based on its CPU queue length. As the CPU queue length drops below its receive threshold value, the processor becomes a receiver and is eligible to accept jobs or tasks. As the CPU queue length increases above its send threshold value, the processor becomes a sender and attempts to transfer jobs or tasks. As for the algorithms selection policy, all jobs or tasks are considered for migration, however, its goal is to avoid preemptive type transfers. As for the algorithm's location policy, either a sender or receiver location policy is followed. The location policy that runs is based on the processors CPU queue length and how it compares to its two threshold values. As for the sender, a suitable receiver is obtained by contacting a processor found in the slightly loaded list and booking a reservation for a job or task to be transferred. As for the receiver, a suitable sender is obtained by contacting a processor found in the heavy-loaded list or ok-loaded list and requesting a job or task transfer. As for the algorithm's information gathering policy, a demand-driven approach is utilized. Information is typically gathered as a processor becomes a sender or receiver. Once a task arrives or departs, information is demanded from other sender or receiver processors to update local lists, and a decision is made as to which location will send and receive the transfer. As for the algorithm's stability control, a great deal is enforced. At a high system load, the sender initiated component becomes disabled. This is due to the fact that the slightly loaded lists on all processors will become empty. At a low system load, the receiver initiated component is somewhat unnecessary. At low system load, the receiver initiated component will almost never obtain an overloaded sender. Since spare CPU power is available during low system load, this receiver polling does not have a negative impact on performance. This algorithm adapts rather nicely to the low and heavy loaded situation. As for the algorithm's local scheduling policy, a first-in-first-out queuing

scheme is usually followed. As figure 5 shows, during task arrival, processor 3, an overloaded (sender), selects moderately loaded (receiver), processor 4, from the Ok list. Processor 4, a moderately loaded (receiver), was found and accepted the task transfer. Meanwhile, during task departure, processor 2, a lightly loaded (receiver), selects overloaded (sender), processor 1, from the High list. Processor 1, an overloaded (sender), was found and subsequently transferred a desired task.

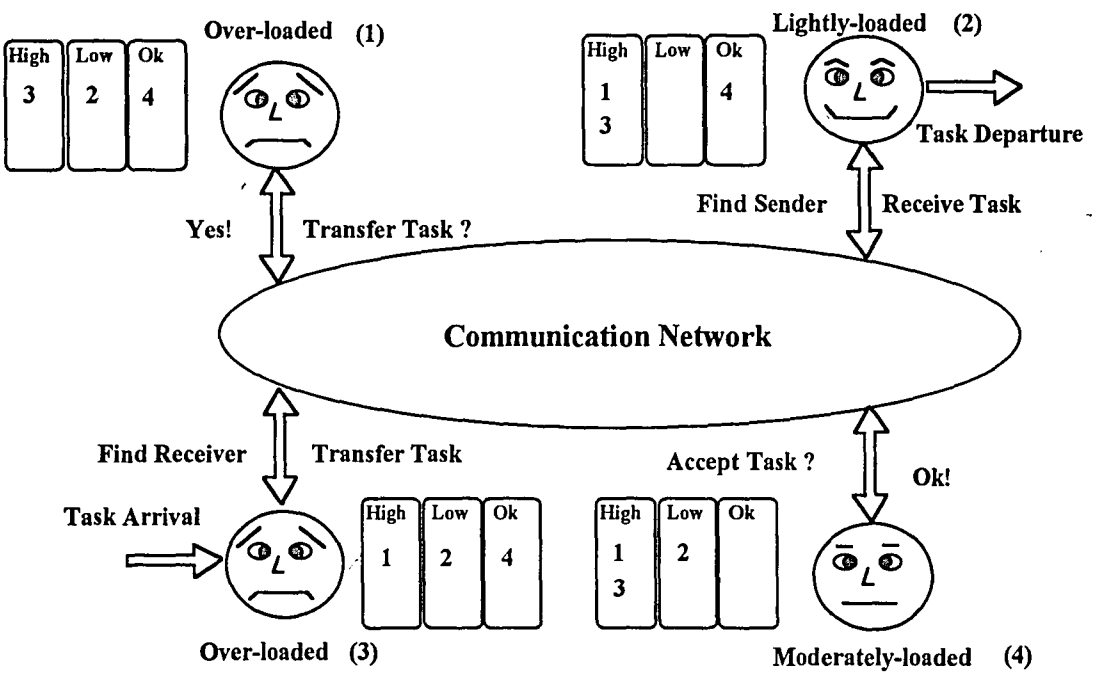

Figure 5 - Adaptive Algorithms

2.6. A Comparison of Algorithms

Refer to table 1 for a comparison of the most predominate dynamic and adaptive distributed job scheduling algorithms (i.e. sender initiated, receiver initiated, symmetrically initiated, and adaptive algorithms). Each algorithm is summarized and compared on the six distributed algorithm policies as well as its performance.

Policy/Algorithm	Sender Initiated	Receiver Initiated	Symmetrically Initiated	Adaptive
Transfer	Job arrival and load index > load threshold	Job departure and load index < load threshold	Job arrival and index > threshold or job departure and index < threshold	Job arrival and index > threshold or job departure and index < threshold
Location	Poll for location where load index < load threshold	Poll for location where load index > load threshold	Poll for location where index < threshold or location where index > threshold	Poll for location where index < threshold or location where index > threshold
Selection	Newly arrived jobs, non-preemptive	Any job that has been requested; non-preemptive or preemptive	Any job that has been requested; non-preemptive or preemptive	Any job that has been requested; non-preemptive or preemptive
Information Gathering	Demand driven on job arrival	Demand driven on job departure	Demand driven on job arrival or departure	Demand driven on job arrival or departure
Stability Control	Not enforced; instability can occur during heavy system loads	Enforced; polls for job transfer cease to exist during heavy system loads	Enforced by receiver but, instability can occur during heavy system loads due to sender	Enforced; receiver lists at processors during heavy system loads are empty
Local Scheduling	First-in-first-out queuing	First-in-first-out queuing	First-in-first-out queuing	First-in-first-out queuing
Performance	Operates well during low and moderate system loads	Operates well during high and moderate system loads	Operates well during low and moderate system loads	Operates extremely well during low, moderate, and high system loads

Table 1 - Algorithm Policy Comparisons

In conclusion, as workstation client/server computing evolves, the sheer processing power attached to a single network will become enormous. In order to harness such raw power and deploy it to its maximum potential, efficient distributed job scheduling algorithms must be researched and developed. As seen in recent literature, the dynamic and adaptive class of distributed job scheduling algorithms are in the forefront. Algorithms such as the sender initiated, receiver initiated, symmetrically initiated, and

adaptive are in practice for they exhibit the ability to react to the overall system load of a distributed system. As we have seen, some algorithms are more efficient and effective than others. Of all the algorithms presented, the adaptive class of algorithm out-performs its counterparts, for it delivers the best performance while maintaining the highest system stability. At a conceptual level, the adaptive class of algorithms is simply a combination of the other classes of algorithms, incorporating their best features. The Distributed Job Management System (DJMS) to be presented, can be classified as an adaptive algorithm for locally networked *UNIX* systems.

Chapter 3. The Distributed Job Management System (DJMS)

In this chapter, the Distributed Job Management System (DJMS) is the main focus. As an introduction, motives and visions for developing DJMS are discussed as well as details about its development and execution platforms. The core of the chapter presents the DJMS operating components. As each of the components are described, comprehensive information regarding their purpose, usage, and interaction is reviewed. In conclusion of this chapter, DJMS's fault tolerance is discussed and a case analysis on system throughput is presented.

3.1. DJMS Motive and Vision

For the most part, the *UNIX* Operating System was designed and developed to support the computing needs of the research, development, engineering, and academic community. Prior to 1986, *UNIX* typically ran on small scale mini and workstation computers supporting a relatively small user base per processor. It wasn't until 1987, that *UNIX* emerged in the business world and began to compete with *MVS* mainframe computing and other proprietary mid-range computing environments[17,19].

In 1988, the AT&T Microelectronics Information System Organization (ISO) began to cautiously migrate applications from a traditional *MVS* mainframe environment to AT&T *UNIX* mini and workstation computers. As business applications were migrated to *UNIX*, key areas of application control and production support, common to the *MVS* mainframe environment, were duly noted as lacking or non-existent. It wasn't until 1989, that some development internal to ISO was undertaken to implement *UNIX* system support tools that would aide the migration efforts. In 1990, research and development began in the development of a local job management system for the *UNIX* Operating

System. This local job management system was the birth of the Distributed Job Management System (DJMS) Vision.

In 1991, a local job management system was deployed into production for *UNIX* background processing. This local job management system provided three basic components or processes. The first component, a user request process, supported background job or task submission for execution. Upon successful completion of the request process, a user job or task would either be in-queue within some job pool or executing on the local processor. The second component, a system level slave monitor daemon process, actually executed and monitored jobs or tasks on the local processor that it obtained from the job pool in a first-in-first-out manner. The third component, a system level master manager daemon process, handled task farming by dictating, at any instance in time, how many slave monitor daemon processes could be simultaneously executed.

This local job management system was very effective for several reasons. First, through the technique of task farming, an automatic throttling mechanism was placed on the processor to prevent applications from overloading the system to the point of thrashing. Second, through the use of several job pools, application jobs could be categorized and queued such that high priority or fast running jobs could be executed without waiting for other larger jobs to complete. Third, by use of the daemon slave monitor process, crucial jobs or tasks could be closely monitored and tracked. In the event of a critical problem, the daemon slave monitor process could trigger the appropriate support. Fourth, by maintaining job or task pools on dedicated storage, the local job management system could re-start and recover application execution from *UNIX* system crashes and failures.

Through 1991 to the present, more and more ISO application have migrated from *MVS* to *UNIX*. During this period of time, numerous *UNIX* mini and workstation

computers have come on-line. The motivation for DJMS has come about as a result of applications requiring more processing power than their local processor can provide. In order to support user response-time requirements, application distribution through means of load sharing and load balancing among computers or processors has become inevitable.

In September of 1992, the DJMS Vision was conceived in the hopes to achieve three specific goals. The DJMS primary goal is to reduce response-time for individual tasks and achieve the highest global processing throughput possible by migrating the execution of jobs to a participating set of non-local, idle or slightly loaded processors. The basic premise is to keep all processors in the network busy, running background jobs, until there are less jobs to execute than the number of participating processors. A secondary goal is to achieve automatic re-start of user background jobs following network and processor failures. Naturally, only the processors that are operable will participate in the background job migrations. A third and final goal is to provide system activity and accounting data as well as statistics for each job submitted and executed. From any of the participating processors, the user must have the ability to backtrack or trace the life of a job within the system. In a simplified view, DJMS has been constructed and developed based on many early concepts found within the ISO local job scheduling system and has incorporated many other techniques found in recent technical literature on distributed job scheduling.

3.2. DJMS Hardware and Software Model

In order to implement DJMS, two homogenous distributed hardware platforms were configured. The first configuration consisted of two AT&T 486 Star Server E and one NCR 3450 server running AT&T *UNIX* SRV4.0. The two Star Server E processors were arbitrarily designated as the slave processors while the NCR 3450 server was deemed the master processor. Each processor was connected to a AT&T *STARLAN 10*

Network. This 10 megabits per second Local Area Network (LAN) was configured as a bus topology and utilized twisted-pair wire at the physical layer. At the data link layer, a typical IEEE 802.3 carrier sense multiple access with collision detection (CSMA/CD) protocol was instituted[16,20]. All user disk space was shared among all three processors through AT&T *Remote File Sharing* (RFS). DJMS was constructed and fully implemented prior to a port to the second hardware configuration.

The second configuration consisted of three Sun Sparc I and two Sun 3/80 workstation running *SunOS* 4.1.3. One of the workstations is arbitrarily deemed the master processor while the others were designated as the slaves. Each processor is connected to Lehigh University's Ethernet Network. This LAN is configured as a bus topology and utilizes coaxial cable at the physical layer. At the data link layer, a typical IEEE 802.3 carrier sense multiple access with collision detection (CSMA/CD) protocol is also instituted[16,20]. All user disk space is shared among all three processors through Sun *Network File Sharing* (NFS).

On top of the mentioned hardware platform, the DJMS software was implemented in the C programming language[17,18,19,20]. The software is physically constructed with one common library which contains 114 sources files where each file contains one function or object. Each of the six DJMS commands are composed of their own specific source files or functions but, rely heavily on common objects from the DJMS library. In all, the DJMS software is composed of 209 source files at a total of 22,097 lines of C code including programmer comments. In general, the DJMS software requires *UNIX* libraries 2, 3C, 3S, and 3N to perform system calls, library calls, and network calls. DJMS interfaces directly into the transport and network layers utilizing the User Data gram Protocol and Internet Protocol (UDP/IP) for processor to processor communications.

DJMS implements its networking through the Berkeley Socket Application Programming Interface (API) or *UNIX* named pipes over RFS.

3.3. DJMS Concepts and Terminology

Prior to presenting the Distributed Job Management System (DJMS), there are terminology and concepts that will be defined and explained. The terms and concepts in table 2 are essentially the basic building blocks for the distributed system model to be presented.

Table 2 - DJMS Concepts and Terminology

- A ***class*** is a logical organization of processes that are assigned to a processor on the network. Processors assigned to a class can usually provide the same processing services. Classes are usually organized with processors that share the same resources, such as I/O peripherals.
- A ***job or task*** is a physical organization of executable code and environmental parameters.
- A ***pool or queue*** is the physical organization on dedicated storage of background job or task requests to be executed. The sequence or ordering for such a structure is usually in a first-in-first-out (fifo) manner.
- A ***master process***, as the name implies, is a supervisor process responsible for maintaining global information in order to accurately control and delegate the flow of processing to its subordinate processes.
- A ***slave process***, as the name implies, is a subordinate process responsible for receiving jobs or tasks from its superior to complete. A subordinate must provide accurate feedback to its supervisor as each task is carried out.

- A ***migration*** is simply the movement of a job or task execution request from one processor to another through the network. As a migration takes place, a job execution request exits a pool from the source processor and is placed into a pool on the destination processor.
- A ***configuration*** is simply a table of information that maps slave processes into a class which directly assigns each slave process to a job pool at a processor on the network.
- A ***transfer policy*** decides when an attempt should be made to migrate a job or task execution request to another processor within a defined class.
- A ***selection policy*** decides which job or task is eligible for a migration. This policy usually dictates whether preemptive or non-preemptive transfers can occur.
- A ***location policy*** decides to which processor a job or task, which is eligible for migration, will be transferred. This policy will attempt to find the optimum destination, i.e., the processor that has the shortest CPU queue length.
- A ***information gathering policy*** decides what information the location policy needs in order to make its decisions and how this information is obtained. The goal is to obtain sufficient information at the least minimum cost in terms of communications and processing. Naturally, if the overhead incurred by gathering the information becomes large, it may offset part or all of the gains obtained by transferring the jobs or tasks.
- A ***stability control policy*** ensures that task are not migrated endlessly. It guarantees that a job will eventually be serviced by a slave process at some participating processor.
- A ***local scheduling policy*** indicates the order in which tasks waiting at a processor will be serviced and executed.
- A ***daemon*** is a process that executes in the background without an associated terminal or login shell. Its sole purpose is to either wait for some event to occur, or wait to perform some specific task on a periodic basis.

- A ***quiesce*** is a systematic and orderly shutdown of either software or hardware components.
- A ***connection less server*** is simply a daemon process that utilizes a message or data gram service to communicate from one system to another over a network.

3.4. DJMS Overview

DJMS is an adaptive load sharing and load balancing system for distributed background processing for processors that utilize the *UNIX* Operating System. DJMS is a stand-alone system external to the *UNIX* Kernel. DJMS is classified as both load sharing and load balancing for it shares and evenly distributes the processing load among its participating cooperative processors. DJMS executes an adaptive algorithm that distributes jobs or tasks based on decisions derived by gathering and maintaining a centralized dedicated store of global system-state information. DJMS assigns tasks to processors by use of a decision making process where peer processors work cooperatively. DJMS is centralized in structure and dictates the placement of jobs or tasks among its slave processors through its job request facility and its centralized master processor. These processes utilize an administratively maintained configuration file and access it over RFS or NFS to determine the current set and state of participating slave processors. DJMS performs jobs or tasks transfers in a non-preemptive manner. Only jobs or tasks that are in-queue and not executing are considered during its transfers or migrations. DJMS utilizes a load index of processor CPU queue length to depict the magnitude of work at each processor.

DJMS supports a number of commands which provide the developer or user with a friendly interface. The Distributed Job Request Facility (djrf) supports user submission of background jobs for execution from any of the participating processors. Based on the processor configuration, the user background job may or may not execute on the

processor in which it was submitted. If the processor in which the job was submitted is already busy, the job most likely will be migrated. Otherwise, the job remains and executes. When a background job does migrate it moves to execute on the next idle or least loaded processor in the set of participating computers. The Distributed Job Cancel Facility (djcf) supports user cancellation and the Distributed Job Status Facility (djsf) supports status of background jobs that have been submitted from any of the participating processors. The Distributed Job Configuration Change Facility (djccf) supports an administration function where the configuration can be dynamically modified. DJMS supports a number of background processes which provide the overall distributed job scheduling infrastructure. The Distributed Job Monitor Facility (djmonf) daemon (the slave process), is a system facility to execute background jobs on one of the participating processors on the network. The Distributed Job Management Facility (djmanf) is a system facility which acts as the master daemon process at one of the participating processors; which is the control processor on the network.

In general, DJMS implements six components or policies that explain the structure of its distributed job scheduling. The first component, its transfer policy, is controlled by its djrf, djmanf, and djmonf processes. The djrf and djmanf processes act as the job or task sender while the djmonf process acts as the job or task receiver. These processes base their decision solely on the comparison of their processor's CPU queue length to the average CPU queue length of all participating processors. A djrf or djmanf process becomes a sender when their processor's CPU queue length raises above the average processor CPU queue length. A djmonf process becomes a receiver when their processor's CPU queue length falls below the average processor CPU queue length of all participating processors. The second component, the selection policy, is controlled by DJMS's djrf and djmanf processes. These processes decide which job or task to actually transfer. Most often newly arriving jobs or tasks will be selected for transfer. Currently,

DJMS does not consider job or task resource requirements such I/O and CPU during its selection. The third component, the location policy, is controlled by DJMS's djrf and djmanf processes. These processes attempt to find the optimum destination, i.e., the processor that has shortest CPU queue length. The fourth component, the information gathering policy, is maintained and controlled by DJMS's djmanf process in cooperation with of all other facilities. A message passing model is utilized as well as network file access to obtain and maintain global-state information. Typically, global-state information is gathered on demand upon a change in the distributed system's state. The fifth component, the stability control policy, is controlled by the djrf and djmanf processes. During heavy distributed system loads, the djrf and djmanf processes temporarily discontinue load distributing activities. As the distributed system load becomes unbalanced, the djrf and djmanf processes re-institute the load distribution. The last and sixth component, its local scheduling policy, is controlled by the djmonf process. At each processor, djmonf executes jobs or tasks in a first in first out manner.

3.5. DJMS Operating Components

The Distributed Job Management System is composed of six components or commands, three of which are the user level interface, while the remaining are strictly for administration and system operation. The Distributed Job Request Facility (djrf) supports user submission of background jobs for execution. The Distributed Job Cancel Facility (djcf) supports user cancellation and the Distributed Job Status Facility (djsf) supports status for background jobs that have been submitted. The Distributed Job Configuration Change Facility (djccf) supports an administration feature where a load sharing configuration can be dynamically modified. The Distributed Job Management Facility (djmanf) and Distributed Job Monitor Facility (djmonf) are daemon processes which

control the overall distributed job scheduling. As each of the components are described, comprehensive information regarding their purpose, usage, and interaction is presented.

3.5.1. Distributed Job Request Facility (djrf)

The Distributed Job Request Facility (djrf) command is a user facility, that can be utilized on any of the DJMS participating processors, to submit background jobs into a class for execution in DJMS. This facility executes as a process but does not execute the user job directly. Based on the supplied class, its processor configuration, and the current system load, it merely transfers the user job into a NFS job pool or queue at some participating processor on the network to be subsequently executed by an appointed Distributed Job Monitor Facility (djmonf) slave process. As figure 6 shows, the djrf process at processor 1 executes the sender initiated portion of the DJMS adaptive job scheduling algorithm. At task arrival, the djrf process performs the selection policy and acts as the job or task sender while the djmonf slave process at processor 2 acts as the job or task receiver. These processes base their transfer and location decisions solely on the comparison of their processor's CPU queue length to the average CPU queue length of all participating processors in the job's class. A djrf process becomes a sender if and only if its local processor's CPU queue length rises above the average processor CPU queue length of all participating processors in the job's class. Otherwise, the djrf process queues the task locally pending the local processor in which the job was submitted is defined within the job's class and exists in the configuration. Once a job has been successfully queued, a UDP/IP message is passed over the network to the Distributed Job Management Facility (djmanf) master daemon process. The message informs the central processor manager, at processor 4, of job execution request. The djmanf master process then ensures that a djmonf slave process is activated or if need be started up on the designated processor to execute the desired user job or task. As the djrf command

completes at processor 1, useful status information about the submitted job is returned to the user. This information, such as a unique job number, can be used later for job status and/or cancellation. This command provides many command line arguments and features. Refer to the appendix for the command manual pages and its documented features.

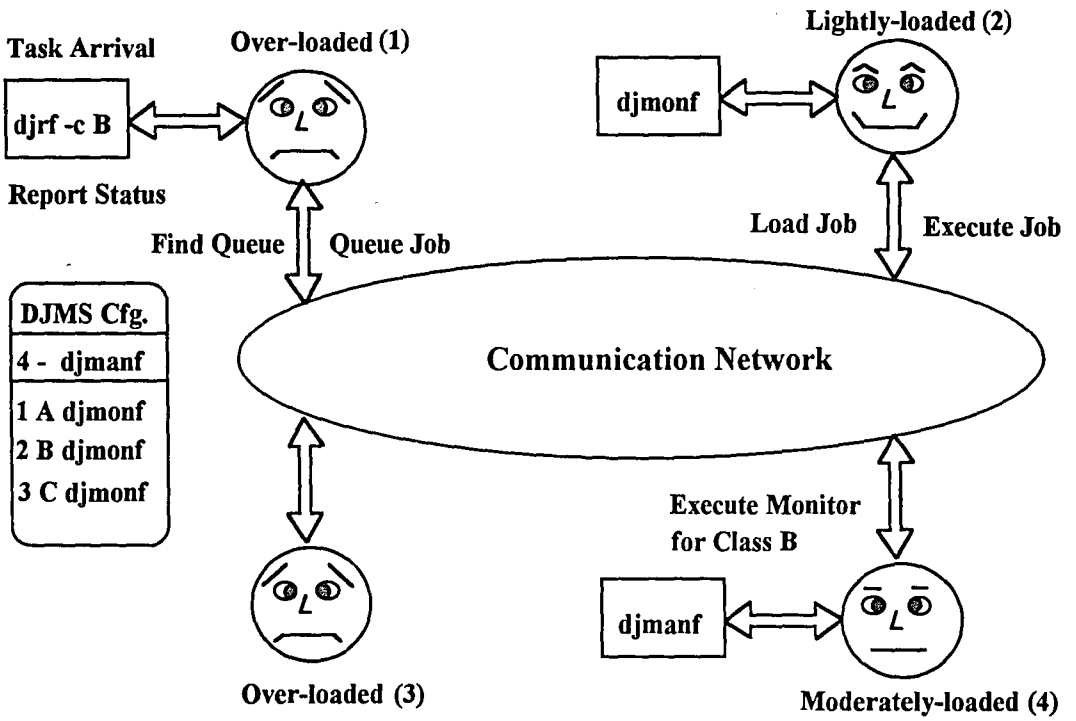

Figure 6 - Job Submission into DJMS

3.5.2. Distributed Job Status Facility (djsf)

The Distributed Job Status Facility (djsf) command is a user facility, that can be utilized on any of the DJMS participating processors, to obtain status of background jobs that have been submitted into DJMS. This facility executes as a process and obtains information on user jobs that may be waiting in a job pool or queue for execution or in execution at participating processors. Upon invoking task status as figure 7 shows, the djsf process at processor 1 accesses job pools or queues through NFS and communicates with participating Distributed Job Monitor (djmonf) slave processes via UDP/IP messages

over the network, in this case one djmonf process at processor 2, to obtain the current DJMS state information. This status facility reports useful run-time information for executing jobs such as current elapsed time and CPU time consumed. The djsf process, at processor 1, also reports information about waiting jobs such as elapsed time in-queue and its order or position in-queue. This command provides a number of command line argument features. Refer to the appendix for the command manual pages and its documented features.

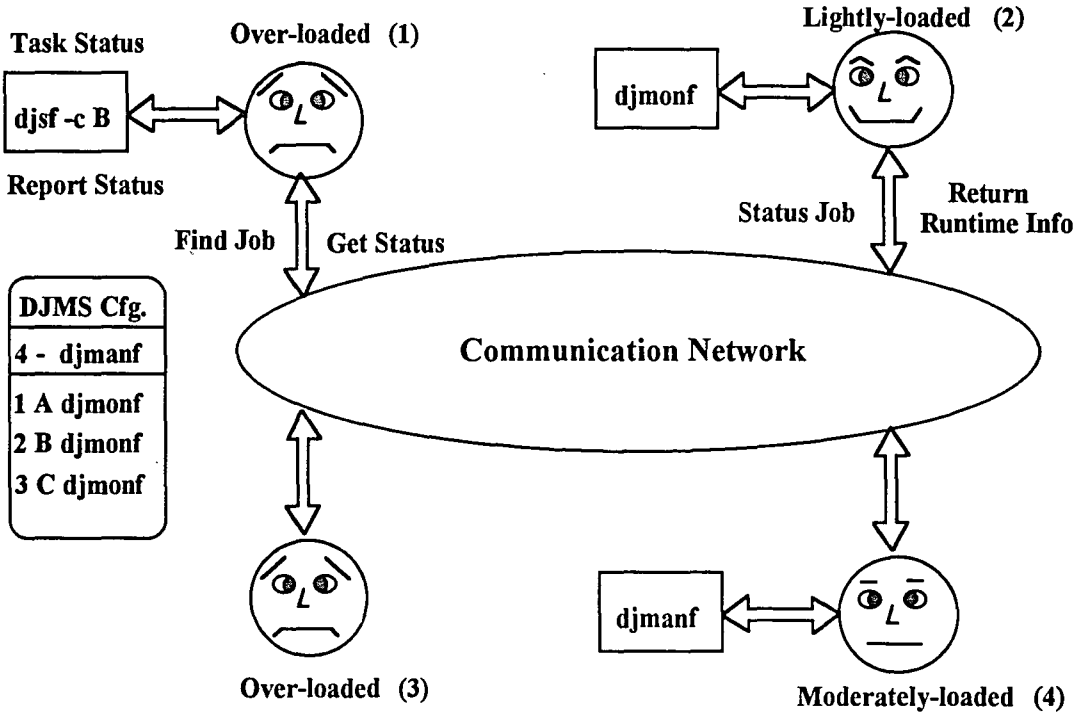

Figure 7 - Job Status and Tracking in DJMS

3.5.3. Distributed Job Cancel Facility (djcf)

The Distributed Job Cancel Facility (djcf) command is a user facility, that can be utilized on any of the DJMS participating processors, to cancel background jobs that have been submitted into DJMS. This facility executes as a process and cancels user jobs that may be waiting in a job pool or queue for execution or in execution at participating

processors. Upon task cancellation, as figure 8 shows, the djcf process at processor 1 accesses job pools or queues through NFS and communicates with participating Distributed Job Monitor (djmonf) slave processes via UDP/IP messages over the network; in this case one djmonf process at processor 2, to cancel user jobs. The djmonf slave process, at processor 2, carries out the actual cancellation. During a cancellation, the djmonf slave process basically kills off the user job's *UNIX* process and physically eliminates the job from the pool or queue. Subsequently, the djmonf slave process reports the cancellation of the user job back to control processor, at processor 4, via a UDP/IP message over the network to inform the Distributed Job Management Facility (djmanf) master daemon process. This command provides a number of command line argument features. Refer to the appendix for the command manual pages and its documented features.

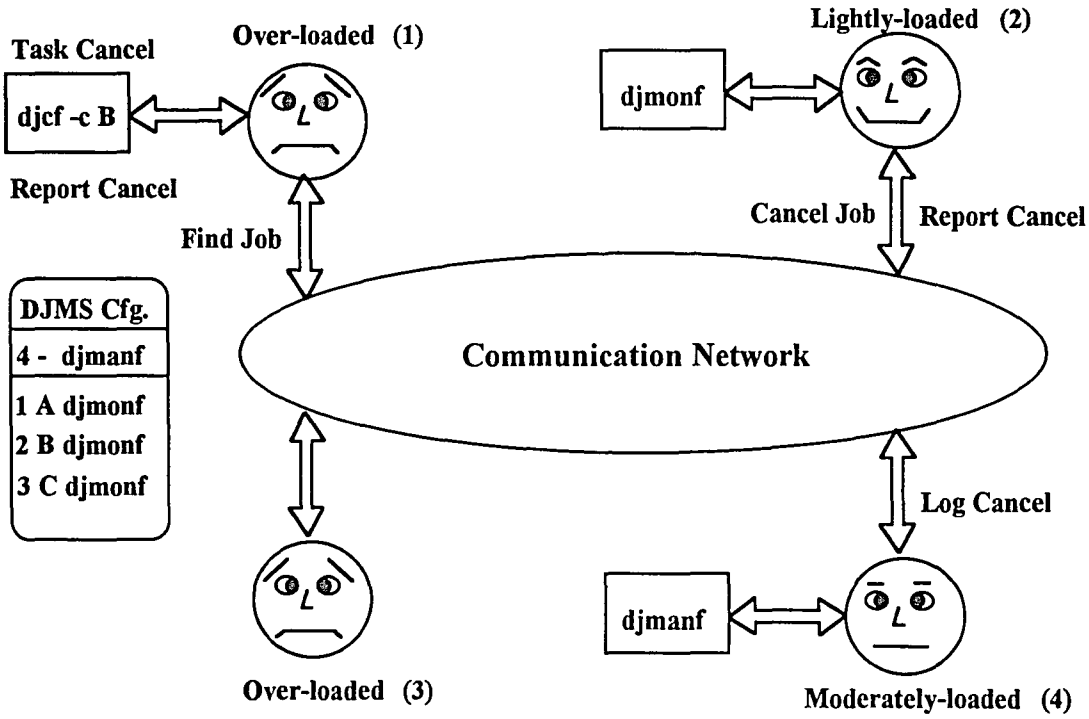

Figure 8 - Job Cancellation in DJMS

3.5.4. Distributed Job Configuration Change Facility (djccf)

The Distributed Job Configuration Change Facility (djccf) command is an administrative change facility used to dynamically configure how many networked processors and concurrent Distributed Job Monitor Facility (djmonf) processes will participate in DJMS. This facility executes as a process and communicates with the master daemon process (the Distributed Job Manager Facility (djmanf)) via UDP/IP data grams over the network to achieve its goal. Basically, it provides the administrator with the ability to tune and adapt the load sharing and load balancing distributed system to better meet user processing needs. It also provides a processing throttle or governor mechanism in that the number of concurrent user background executions across the participating processors can be controlled efficiently and automatically. Configuration changes can be performed either interactively or in the background. During a configuration change as figure 9 shows, the djccf process, at processor 1, has the capability to either add, modify, or delete class and processor attributes for participating slave processors as well as the master processor. As a configuration change takes place, updates through NFS or RFS are performed to dedicated storage for all affected pools or queues as well as the configuration file itself. Next, the Distributed Job Management Facility (djmanf) master daemon process, at processor 4, is notified via messages to perform the sender initiated portion of the DJMS algorithm. As jobs and locations are identified, the djmanf process dispatches the necessary migrations while initiating the necessary Distributed Job Monitor Facilities (djmonf) slave daemon processes. As in this case, at processor 2, the djmonf process is initiated to institute the receiver initiated portion of the DJMS algorithm. After the djmonf slave daemon process starts up, it loads and executes jobs or tasks obtained from its pool or queue. This command provides a number of command line argument features. Refer to the appendix for the command manual pages and its documented features.

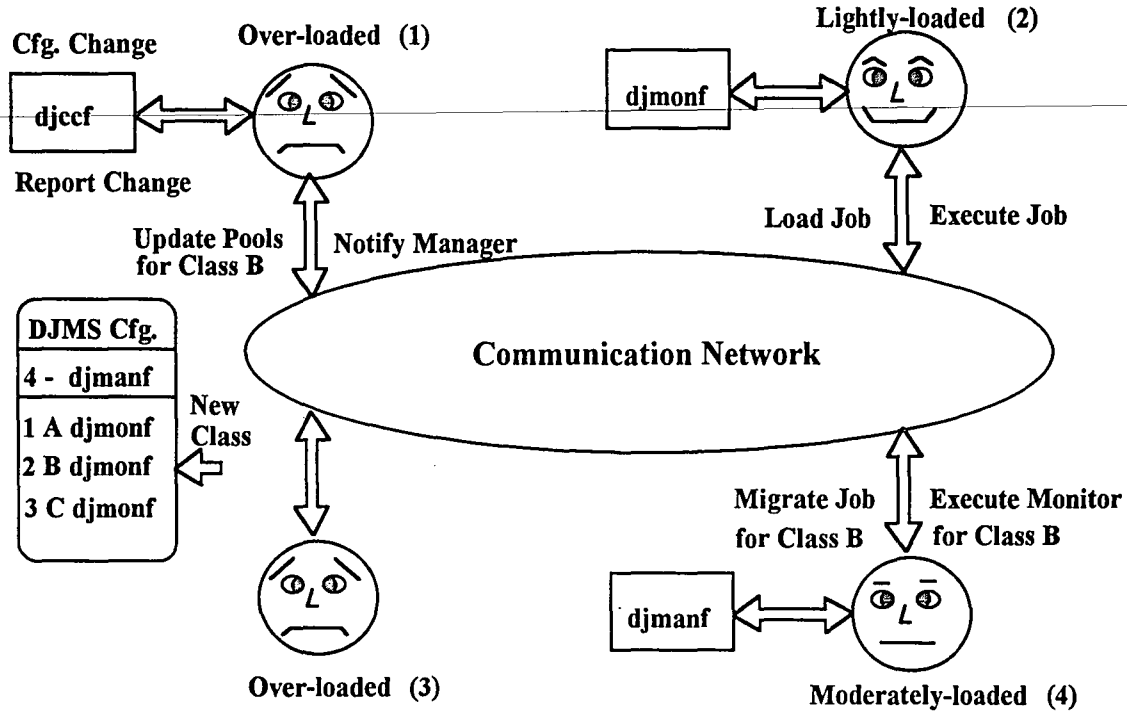

Figure 9 - Configuration Change in DJMS

3.5.5. Distributed Job Monitor Facility (djmonf)

The Distributed Job Monitor Facility (djmonf) daemon, the slave process, is a system facility to execute background jobs on a participating processors on the network. This facility may execute as a connection less server process on each processor defined in the configuration. The sole purpose of djmonf is to run the receiver initiated portion of the DJMS algorithm and also execute, monitor, terminate, and report information about user background jobs. It will also ensure automatic re-start of user background jobs after processor or network failures. This slave facility communicates, synchronizes, and cooperates with its one and only master Distributed Job Management Facility (djmanf) process at the control processor. The local scheduling policy is strictly managed by the djmonf process at each processor. As figure 10 shows, the djmonf process, at processor 2, services jobs or tasks in a first-in-first-out (fifo) manner where both priority of the job

and elapsed wait time in-queue have bearing. On demand by the Distributed Job Management Facility (djmanf), at processor 4, djmonf will be signaled or remotely spawned on a participating processor, in this case processor 2, to schedule a particular job or task request. Once djmonf has started, each job request contained in its assigned input job pool is processed. Each job or task is executed and monitored. If the job or task fails, subsequent executions may take place, but not necessarily on the same processor. During each job execution and after each successful job completion or failure, pertinent information is transferred over the network via UDP/IP data grams to djmanf daemon in order to update the global DJMS state and possibly trigger subsequent job migrations. After all jobs have been completed and the designated input job pool is empty, djmonf will message djmanf, at processor 4, of its termination and will terminate. On demand, djmonf at processor 1 can be signaled to quiesce or cancel a distributed job. The djmonf command (if executing) will signal the current job command *UNIX* process to terminate. Once the job or task has terminated at processor 1, djmonf will message djmanf of its termination. During a quiesce, djmonf will shutdown immediately and terminate.

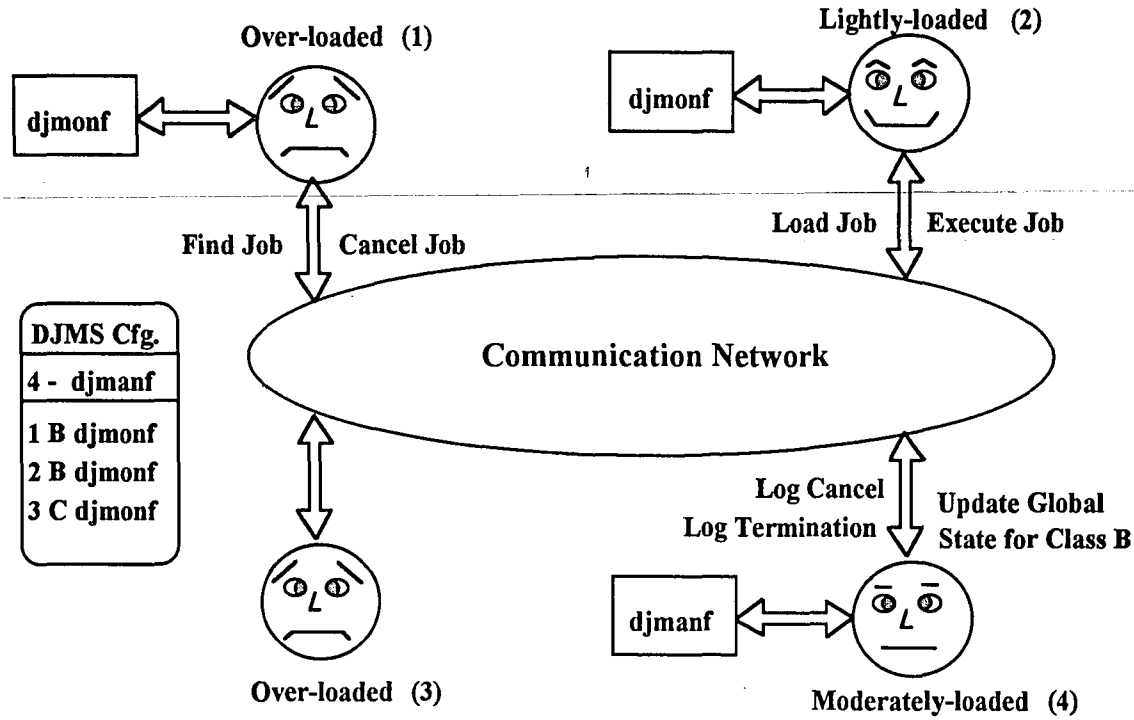

Figure 10 - Load Sharing in DJMS

3.5.6. The Distributed Job Management Facility (djmanf)

The Distributed Job Management Facility (djmanf) is a system facility, a connection less server which acts as the master daemon process at one of the networked processors; the control processor on the network. Its sole purpose is to perform the sender initiated and adaptive portions of the DJMS algorithm. In doing so, it executes, monitors, and terminates its slaves, the Distributed Job Monitor Facility (djmonf) processes, while maintaining the global-state of DJMS. By gathering information from each of its slave processes executing on each of the participating processors, djmanf records and maintains the global-state in its control processor's shared memory segment as well as on dedicated networked disk storage. Based on this global information, the Distributed Job Manager process deploys its load sharing policies. It will also ensure automatic re-start of failed slave processes after processor or network failures. It supports a single point of control for DJMS initiation and termination. This master facility will

communicate and synchronize itself with each of its djmonf slaves. The Distributed Job Manager process manages a transfer, selection, location, information gathering, and stability control set of policies. The transfer policy is based solely on the comparison of each participating processor's CPU queue length to the average CPU queue length of all participating processors within a designated class. Very simply, jobs are migrated between job pools within a class to share and balance the processing load. During a migration, queued jobs may shift from one processor to an other until executed. The selection policy is based solely on identifying jobs or tasks that are not presently executing on a processor. Again, DJMS only supports non-preemptive transfers. The location policy is based solely on the pre-defined DJMS configuration as well as CPU queue lengths. This configuration dictates for a given job class how many jobs can execute concurrently as well as on which participating processors on the network. The information gathering policy is managed by two methods. First, a message passing model exists where the master process at the control processor communicates via UDP/IP data grams over the network to all its slave processes at participating processors. Second, the master process on a event trigger basis polls each slave's job pool for CPU queue length information. Through these information gathering techniques, the master process has accurate global knowledge as to what is occurring in the distributed system. As figure 11 shows, stability control is ensured cooperatively by the master djmanf process at processor 4 and the Distributed Job Request Facility (djrf) process, in this case at processor 1. As the distributed system load becomes increasingly heavy, the master djmanf process and the djrf processes ensure that for each class all participating slave processors have evenly dispersed CPU queue lengths. As long as the jobs or tasks are evenly dispersed among processors, the sender initiated transfer policy becomes dormant. In general, DJMS minimizes the inefficiencies of endless sender initiated transfers under heavy system loads. At heavy system loads, transfer or migration are solely triggered by a djmonf, at processor 2, running the receiver initiated portion of the DJMS algorithm as jobs or tasks complete and depart. On demand by the

Distributed Job Request Facility (djrf) process or periodically, djmanf will awaken via a receipt of UDP/IP message to schedule Distributed Job Monitor Facility (djmonf) processes. Once djmanf has awakened, all defined configuration job pools are polled through NFS or RFS for the presence of job requests. Each configured djmonf process whose job pool contains job requests will be executed (if not already executing). After all necessary configuration job monitor scheduling has been completed successfully, djmanf will listen for subsequent events. On demand by receipt of a djmonf UDP/IP data gram, djmanf will awaken to recognize that the particular Distributed Job Monitor Facility (djmonf) has started or terminated. On demand by a specific djmonf process, djmanf will awaken to gather global DJMS state information. Once djmanf has awakened and recognized that a Distributed Job Monitor has terminated or a job request has completed, djmanf will again poll through NFS or RFS the job pools for the presence of distributed job requests. Again distributed monitor scheduling may take place after all load balancing via job migration has completed. The djmanf process will again listen for subsequent events. On demand, djmanf could be awakened to quiesce all executing Distributed Job Monitor Facilities. Once djmanf has awakened and recognized the quiesce, djmanf will immediately ignore any incoming requests from djrf processes and will signal via a *UNIX* kill system call each running Distributed Job Monitor (djmonf) process to terminate. Once all Distributed Job Monitor (djmonf) processes have reported their termination, djmanf will terminate as well.

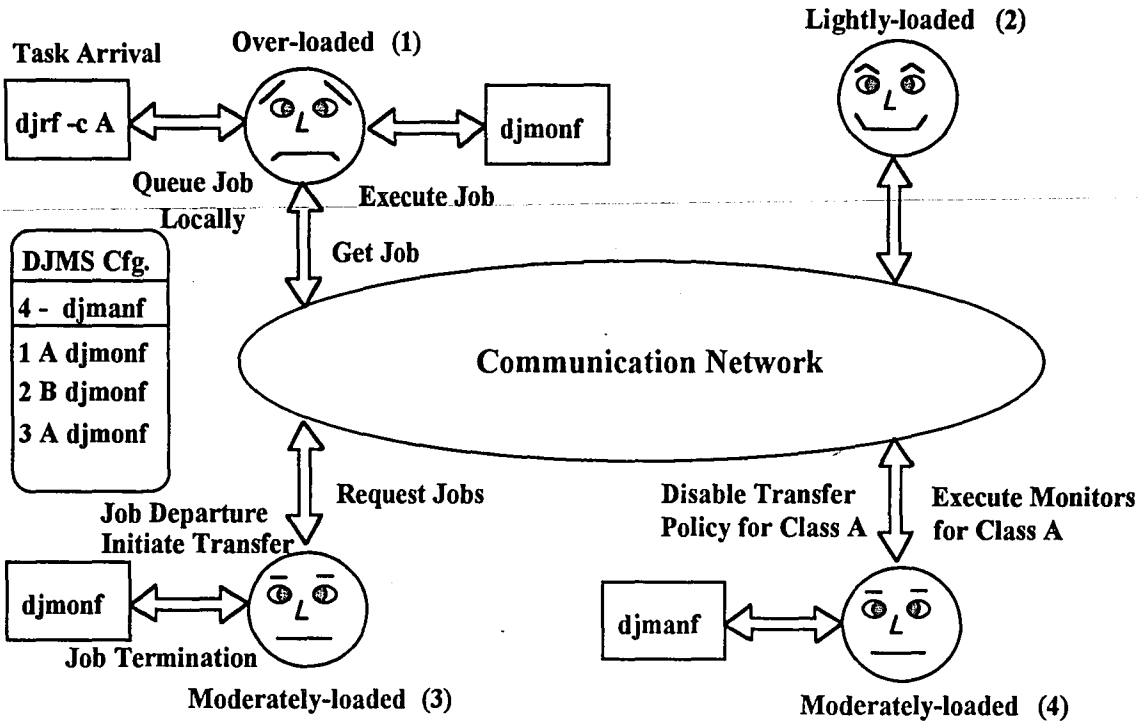

Figure 11 - Stability Control in DJMS

3.6. A Summary of the DJMS Components

Refer to table 3 for a summary of the DJMS Components (i.e. Distributed Job Request Facility (djrf), Distributed Job Status Facility (djsf), Distributed Job Cancel Facility (djcf), Distributed Job Configuration Change Facility (djccf), Distributed Job Monitor Facility (djmonf), and Distributed Job Management Facility (djmanf)). Each component is summarized and compared on the six distributed algorithm policies as well as their usage and purpose.

Policy/ DJMS	djrf	djsf	djcf	djccf	djmonf	djmanf
Transfer	Job arrival and cpu queue length > avg. cpu queue length	None	None	None	Job departure and cpu queue length < avg. cpu queue length	Job arrival and cpu queue length > avg. cpu queue length
Location	Processor with shortest cpu queue length	None	None	None	None	Processor with shortest cpu queue length
Selection	Newly arrived jobs; non-preemptive	None	None	None	None	Newly arrived jobs; non-preemptive
Information Gathering	Demand driven on job arrival	None	None	None	None	Demand driven on job arrival and departure
Stability Control	Sender enforced; transfers become disabled	None	None	None	None	Sender enforced; transfers become disabled
Local Scheduling	None	None	None	None	First-in-first-out queuing	None
Usage	User	User	User	Admin.	System	System
Purpose	Submit jobs	Status jobs	Cancel jobs	Change config.	Execute and monitor jobs	Control and monitor environment

Table 3 - A Summary of the DJMS Components

3.7. DJMS Fault Tolerance

The Distributed Job Management System (DJMS) was designed and developed with rigid fault tolerance, most of which is accomplished by the Distributed Job Management and Monitor Facilities. For the most part, DJMS's fault tolerance overcomes the critical disasters that could be encountered in the case of a master processor failure. From a pure academic point of view though, a single point of control is not usually desirable. Single points of control can lead to load distribution bottlenecks as well as one point of failure. As figure 12 shows, at control processor 4, the Distributed Job

Management Facility (djmanf) frequently polls for off-line or down participating processors as well as dead Distributed Job Monitor Facilities (djmonf) processes on each of the participating processors. The djmanf process makes periodic attempts to re-start dead Distributed Job Monitor Facilities, in this case processor 2, upon slave processor recovery. As participating processors on the network go down, jobs or tasks migrate automatically to other peer processors as defined by the configuration. This ensures the automatic re-start of user jobs or tasks that may have been executing during a processor crash. It also ensures that jobs or tasks in-queue for a crashed processor will migrate to other available processors. The Distributed Job Management Facility (djmanf) utilizes local shared memory as well as network shared dedicated storage to keep track of all global-state information. In the case of a control processor failure, all Distributed Job Monitor Facilities (djmonf) on each participating processor suspend execution and wait for a tunable period of time for the Distributed Job Management Facility (djmanf) process to recover. Upon control processor recovery, the Distributed Job Manager re-gains its global knowledge from its local shared memory segment and shared dedicated storage. Once the DJMS global-state has been restored, synchronization takes place between the djmanf master process and each of its slave djmonf processes. In the event that the control processor never recovers, the DJMS administrator can define a new control processor via the Distributed Job Configuration Change Facility (djccf). By defining a new master processor in the DJMS configuration, a new instance of the Distributed Job Management Facility (djmanf) process will be started to re-gain DJMS global system state from the shared dedicated storage. Once global-state has been restored, synchronization takes place between the djmanf master process and each of its slave djmonf processes. In the event of a failure on the shared dedicated storage medium, DJMS will immediately quiesce. From a networking point of view, DJMS has implemented a rather robust message transfer service. For the most part though, the User Data gram Protocol (UDP) message services utilized is not reliable[19]. UDP by design does not ensure that the

message transmitted from the source made it to the destination. As we know from experience due to invalid packet checksums, messages will get lost in a network. In order to compensate from some of its deficiencies and make the message service reliable, four features were developed. First during data gram creation, a sequence number was added. Second during a data gram send, if an acknowledgment from the destination is not received the data gram will be re-transmitted. The re-transmission will occur for a tunable number of times. Third during a data gram receive, the data grams sequence number is interrogated to determine whether the data gram received is a duplicate, out of sequence, or appropriate. Fourth and last during a data gram receive, an acknowledgment is returned to the originator.

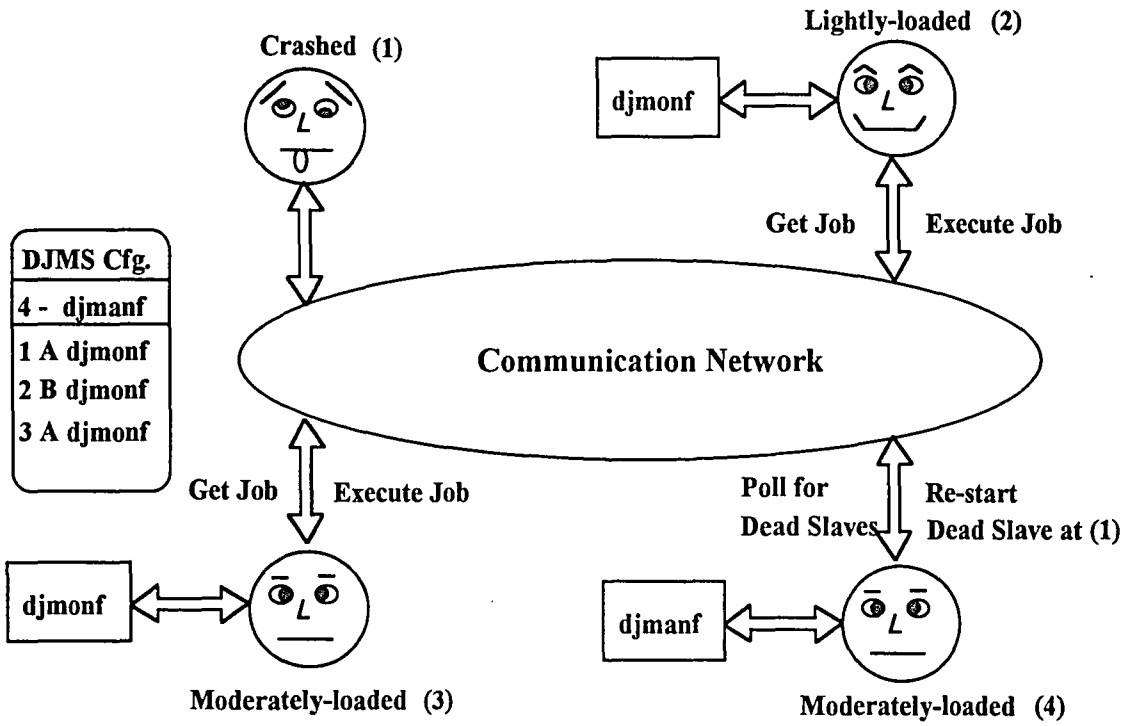

Figure 12 - Fault Tolerance in DJMS

3.8. DJMS Throughput Analysis

Often the main goal of distributed load sharing and load balancing is to minimize the overall average job or task response-time[1,2,3]. A Job or task response-time can simply be defined as the job's elapsed run-time (i.e. job's completion time less its start time). In distributed systems, average response-time has been widely used as a metric for performance measurements. In order to analyze system throughput and performance in DJMS, numerous bench marks or tests were generated and studied within the *SunOs* 4.1.3 environment.

First module *compute*, a simple C program, was constructed as the user job or task for DJMS bench marking purposes. The *compute* program was structured with a large for-loop where its loop index runs from 1 to 10,000,000. Within each iteration of *compute's* loop, its index is multiplied and assigned to another variable. With such a CPU bound job like *compute* in place, different bench marks or tests were instituted.

In total, five specific bench marks or tests were performed. Common to each bench mark was 50 concurrent requests to execute *compute*. Each of these requests were submitted into DJMS via the Distributed Job Request Facility (djrf) with the same class (e.g. class A) parameter. In each test, the DJMS configuration's class A information was defined to permit, at any point in time, a maximum of 10 simultaneous user job executions (e.g. *compute* jobs). Unique to each bench mark, the DJMS configuration's class A information was modified to add or remove participating processors. Benchmarks were performed with 1 to 5 Sun processors. As each bench mark was run, DJMS's affects on job response-time and total system throughput were captured and analyzed. In all cases while these experiments were performed, the participating processors may have responded to other unrelated jobs in addition to the jobs executed for the tests. Hence, larger improvements in both job response-time and system throughput could have possibly been

achieved. In studying the resulting response-time data, significant improvements are observed. First in-terms of system throughput, the elapsed time to complete all 50 *compute* jobs with 1 participating processor was 1,457 seconds versus 610 seconds to complete all 50 *compute* jobs using 5 processors. Thus, the 1 processor configuration took roughly 2.4 times longer than the 5 processor configuration to complete its work. Second in-terms of job response-time, the average response-time to complete all 50 *compute* jobs with 1 participating processor was 239 seconds versa 83 seconds to complete all 50 *compute* jobs using 5 processors. Thus, the 1 processor configuration took roughly 2.9 times longer on the average to complete a *compute* job.

In summary, as more processors participated in DJMS's load sharing and load balancing activities (1) job response-times decreased while (2) system throughput increased. Refer to table 4 for a summary of each of the 5 bench marks and their results.

Benchmark	Jobs	Sun Processors using *SunOs* 4.1.3	Time (secs.) to complete all jobs	Minimum response-time (secs.)	Maximum response-time (secs.)	Average response-time (secs.)
1	50	(1) lion	1,457	119	276	239
2	50	(2) lion, pluto	826	80	238	119
3	50	(3) lion, pluto, mars	787	60	206	114
4	50	(4) lion, pluto, mars, saturn	679	56	191	96
5	50	(5) lion, pluto, mars, saturn, neptune	610	54	176	83

Table 4 - Summary of the DJMS Throughput Analysis

Refer to figure 13 for a bar chart of each of the 5 bench marks. In general, the chart provides a visual comparison with pattern filled bars representing the elapsed, maximum, minimum, and average response-times for each of the 5 tested DJMS configurations. After the results for each configured bench mark was examined, it becomes obvious that, as processors were added to the distributed system, the overall global throughput had increased.

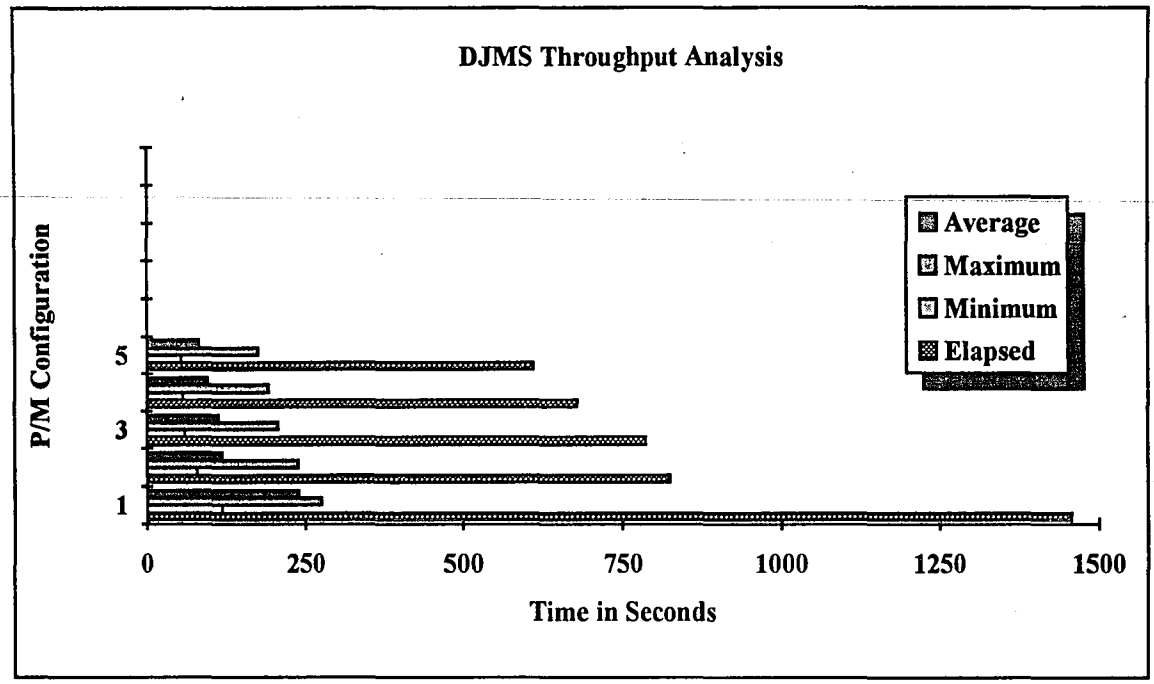

Figure 13 - DJMS Throughput Analysis

Refer to figure 14 for a line graph of each of the 5 bench marks. In the graph legend, letter **P** refers to processor while letter **M** refers to Distributed Job Monitor Facility (djmonf). As an example **1 P / 10 M** represents a DJMS configuration of 1 processor that, at any point in time, permits a maximum of 10 simultaneous djmonf and user job processes. In the graph, arrow label **Load Sharing Peak** depicts where in the DJMS bench marks load distribution peaked while arrow label **Load Sharing Decline** depicts where in the DJMS bench marks load distribution declined. Arrow label **No Load Sharing** simply emphasizes the line in which no DJMS load distribution occurred. In general, each line within the graph plots the actual response-time for each the 50 *compute* jobs submitted and executed in DJMS under the 5 different configurations. As each of the bench marks were performed, job response-times decreased as more participating processors were added.

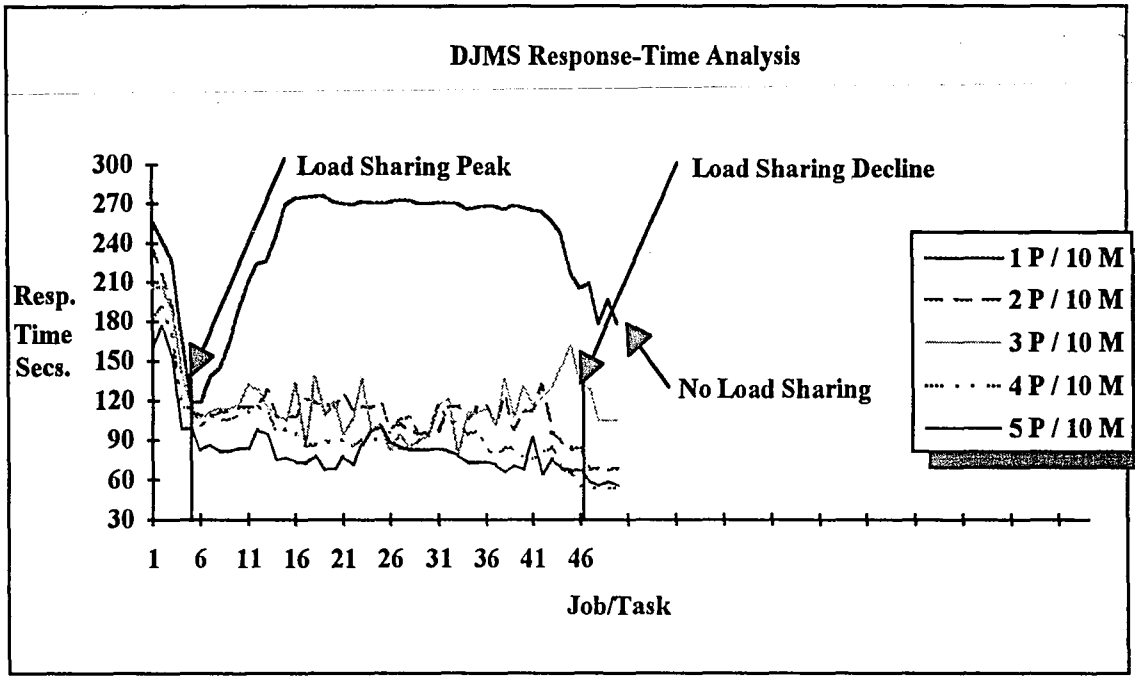

Figure 14 - DJMS Response-Time Analysis

Chapter 4. Conclusion and Future Work

With the availability of high speed local area networks and powerful workstations and mini-computers, both business and academic computing have begun to migrate from the traditional and centralized mainframe to a decentralized and distributed processing environment. With proper load distribution through distributed job scheduling algorithms, jobs or tasks can transparently migrate and subsequently execute at neighboring processors with best intentions to (1) maximize global system throughput and (2) meet job or task response-time requirements.

Distributed job scheduling algorithms today can be classified as either static, dynamic, or adaptive and are either centralized, decentralized, or hierarchical in structure. By far the adaptive class of algorithms out-perform their counterparts. During distribution of jobs or tasks, the transfer can be either be preemptive or non-preemptive. Preemptive type transfers are typically complicated and expensive. In all distributed scheduling algorithms, a key component to its success lies within the effectiveness of its load index. Tracking CPU queue lengths as a load index at participating processors is common. In the current technical literature, most distributed job scheduling algorithms are described and explained through six basic components: the transfer policy, selection policy, location policy, information gathering policy, stability control policy, and local scheduling policy. A large portion of this literature concentrates on the explanation and study of predominate dynamic and adaptive distributed algorithms (i.e. the sender initiated, receiver initiated, symmetrically initiated, and adaptive algorithms).

The Distributed Job Management System (DJMS) is a centralized distributed job scheduling system that performs adaptive load sharing policies and load balancing activities for background processing on locally networked *UNIX* processors. The DJMS primary goal is to reduce response-time for individual tasks and achieve the highest global

processing throughput possible by migrating the execution of jobs to a participating set of non-local, idle or slightly loaded processors. A secondary goal is to achieve automatic re-start of user background jobs following network and processor failures. A third and final goal is to provide system activity and accounting data as well as statistics for each job submitted and executed. DJMS is composed of six components or commands. The Distributed Job Request Facility (djrf), the Distributed Job Cancel Facility (djcf), and the Distributed Job Status Facility (djsf) are the user level interfaces. The remaining commands, the Distributed Job Configuration Change Facility (djccf), the Distributed Job Management Facility (djmanf), and the Distributed Job Monitor Facility (djmonf) are strictly for administration and system operation. The DJMS software was implemented in the C programming language and has two ports available under AT&T *UNIX* SRV 4.0 and Sun *SunOs* 4.1.3. The Distributed Job Management System (DJMS) was designed and developed with rigid fault tolerance, most of which is accomplished by the Distributed Job Management and Monitor Facilities. In-terms of performance, DJMS bench marks have proven that with proper load distribution of jobs among networked processors, job or task response-time decreases while system throughput increases. In order to fully quantify DJMS's benefits, future *UNIX* pilot projects and applications must extensively deploy DJMS services. Furthermore, future research and development efforts must be pursued to eventually market the Distributed Job Management System as a commercially available *UNIX* product. In conclusion, software utilities such as DJMS must become available in order to harness the ultimate power of networked computers and provide a seamless environment for distributed systems.

List of References

[1] M. Schaar, K. Efe, L. Delcambre, and L. N. Bhuyan, "Load Balancing with Network Cooperation," *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991, pp. 328-334.

[2] N. G. Shivaratri and M. Singhal, "A Transfer Policy for Global Scheduling Algorithms to Schedule Tasks With Deadlines," *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991, pp. 248-255.

[3] N. G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992, pp. 33-44.

[4] S. Yuan, "An Efficient Periodically-Exchanged Dynamic Load Balancing Algorithm," *Proceedings of the ISMM International Conference*, 1990, pp. 149-153.

[5] H. C. Lin and C. S. Raghavendra, "A Dynamic Load Balancing Policy with a Central Job Dispatcher (LBC)," *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991, pp. 264-271.

[6] K. G. Shin and C. J. Hou, "Effective Load Sharing in Distributed Real-Time Systems," *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, 1991, pp. 670-677.

[7] K. G. Shin and Y. Chang, "Load Sharing in Distributed Real-Time Systems with State-Change Broadcasts," *IEEE Transactions on Computers*, Vol. 38, No. 8, Aug. 1989, pp. 1124-1142.

[8] K. Ramamrithm, J. A. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Transactions on Computers*, Vol. 38, No. 8, Aug. 1989, pp. 1110-1123.

[9] T. Kunz, "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme," *IEEE Transactions. Software Engineering.*, Vol. 17, No. 7, July 1991, pp. 725-730.

[10] Y. Artsy and R. Finkel, "Designing a Process Migration Facility: The Charlotte Experience," *Computer*, Vol. 22, No. 9, Sept. 1989, pp. 47-56.

[11] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software Practice and Experience*, Vol. 21, No. 8, Aug. 1991, pp. 757-785.

[12] N. G. Shivaratri and P. Krueger, "Two Adaptive Location Policies for Global Scheduling," *Proceedings of the 10th International Conference on Distributed Computing Systems*, 1990, pp. 502-509.

[13] P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991, pp. 336-343.

[14] T. L. Casavant and J. G. Kuhl, "Effects of Response and Stability on Scheduling in Distributed Computing Systems, " *IEEE Transaction. Software Engineering.*, Vol. 14, No. 11, Nov. 1988, pp. 1,578-1,587.

[15] V. M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Transactions on Computers*, Vol. 37. No. 11, Nov. 1988, pp. 1384-1397.

[16] S. S. Lam, "A carrier sense multiple access protocol for local networks," *Computer Networks*, Vol. 4, 1980, pp. 21-32.

[17] B. W. Kernighan and D. M. Ritchie, *The C Programming Language 2nd. Ed.*, Prentice-Hall, Inc., 1988.

[18] D. E. Comer, *Internetworking With TCP/IP, Second Edition, Volume I*, Prentice-Hall, Inc., 1991.

[19] W. R. Stevens, *UNIX Network Programming*, Prentice-Hall, Inc., 1990.

[20] R. D. Rosner, *Distributed Telecommunications Networks via Satellites and Packet Switching*, Lifetime Learning Publications, 1982.

Appendix

DJRF(1) **(Distributed Job Management Facilities)** **DJRF(1)**

NAME

djrf - Distributed Job Request Facility

SYNOPSIS

djrf [-v]
[-m]
[-a]
[-A]
[-n name]
[-g group]
[-u user]
[-c class]
[-P priority]
[-T elapsed run-time]
[-C CPU run-time]
[-s cancel signal]
[-U disk usage limit]
[-L log output file]
[-Q execution retries]
[-e command executable]

DESCRIPTION

The Distributed Job Request Facility (djrf) command is a user facility, that can be utilized on any of the DJMS participating processors, to submit background jobs into a class for execution in DJMS. This facility executes as a process but does not execute the user job directly. Based on the supplied class, its processor configuration, and the current system load, it merely transfers the user job into a NFS job pool or queue at some participating processor on the network to be subsequently executed by an appointed Distributed Job Monitor Facility (djmonf) slave process.

The parameters to djrf are:

-v

Verify all supplied arguments without queuing the actual distributed job request. This option will echo all options and arguments supplied. It will also echo all notes, warnings and errors detected.

-m

Send invoking user or a specified user mail upon completion of the distributed job request. Note, if errors and/or warnings are encountered while processing a particular request, the invoking user will receive mail regardless if this option is supplied or not. *UNIX* mail(1) must be installed for this option to be supported.

-a

Interface to the emergency support system upon distributed job failures. Note, if no failures are encountered while processing a particular request or this option is not supplied, no alerts will be generated. The emergency support system alert(1) must be installed for this option to be supported.

-A

Provide job accounting information for the current request. If the option is supplied, the following information is reported for all job step/processes: start time, end time, elapsed run time, CPU time, characters transferred, block reads/writes. *UNIX* acctcom(1) must be installed for this option to be supported.

-n name

The job name or identification for the current request. This option allows the user to attach a unique id to each distributed job request for ease of future cancellation or status purposes. If the option is not supplied, job name will default to the invoking user's environment variable LOGNAME. The name is limited to 8 characters in length.

-g group

The job group for the current request. This option allows the user to attach a unique id to a group of distributed job request for ease of future cancellation or status purposes. If the option is not supplied, job group will default to the supplied or assigned class. The group is limited to 8 characters in length.

-u user

The user/owner login name or electronic mail address to receive mail. If the option is not supplied, user will default to the current value of defined environment variable LOGNAME.

-c class

The execution class. Based on the class supplied, the Distributed Job Management Facility (djmanf) determines which Distributed Job Monitor Facility (djmonf) on a designated processor will receive the request. The

current class and pool configuration can always be obtained by viewing the file assigned to #define CFG in include file djms.h.

-P priority

The job execution priority within the supplied class. The priority value can range from 100 to 109 where the lower the value the higher the priority. If the argument is not supplied the default priority is 109.

-T elapsed run-time

The maximum amount of time given to the specified executable to run. Once the process has exceeded the elapsed time, it will be killed. If the argument is not specified, by default elapsed time will be obtained from environmental variable RUNTIME or will default to 00:00:00 as an unlimited amount.

-C CPU run-time

The maximum amount of CPU time given to the specified executable to run. Once the process has exceeded the cpu time allocated, it will be killed. If the argument is not specified, by default CPU time will be obtained from environmental variable CPUTIME or will default to 00:00 as an unlimited amount.

-s cancel signal

The signal number to be used by the Distributed Job Monitor Facility (djmonf) upon user or system cancellation. If the argument is not specified, by default cancel signal will be set to SIGKILL (sure kill signal 9).

-U disk usage limit

The maximum number of 512 disk blocks that could possibly be allocated for writing to any particular output files. If the argument is not specified, by default the current environmental user limit will be utilized.

-L log output file

The full path to the log output file that will be created by the command executable stdout and/or stderr messages. If a full path is not supplied, environmental variable LFPATH or PWD will be utilized to obtain the full path to the log output file. If the argument is not supplied, all stdout and/or stderr messages will default to log output file #define SAR/job-id, where SAR is defined in include file djms.h and job-id is comprised of a class, slash (/), name, period (.), and some unique number.

-Q execution retries

The maximum number of execution retries for a job request. If execution retries is greater than zero and a job execution fails, the job is not immediately re-run. Instead the job request is re-queued with execution retries minus one and the next lowest priority. Eventually, the request will be re-scheduled on some processor. If the argument is not specified, by default zero retries will be assumed.

-e command executable

The full path to the command that will be executed and monitored by its parent Distributed Job Monitor Facility (djmonf). If a full path is not supplied, environmental variable EFPATH or PWD will be utilized to obtain the full path to the command executable.

EXAMPLES

Run command /usr/bin/cat -s /etc/motd as job EXP_JOB in group EXP_GRP with a maximum elapsed run time of twenty seconds and a maximum cpu time of two seconds with the highest priority within the B execution class.

```
djrf -n EXP_JOB -g EXP_GRP -c B -P 100 -T 00:00:20 -C 00:02 -e "/usr/bin/cat
-s /etc/motd"

11/20/93 12:34:02 djrf   : B-EXP_JOB.12342 EXP_GRP 100B.1031 queued

RUNTIME=00:20:00; export RUNTIME
CPUTIME=00:02; export CPUTIME
EFPATH=/usr/bin; export EFPATH

djrf -n EXP_JOB -g EXP_GRP -c B -P 100 -e "cat -s /etc/motd"

11/20/93 12:34:02 djrf   : B-EXP_JOB.12342 EXP_GRP 100B.1031 queued
```

FILES

/usr/djms/adm/cfg
/usr/djms/adm/log
/usr/djms/adm/lock
/usr/djms/adm/sar/*/*
/usr/djms/pool/*/*

SEE ALSO

djsf(1),djcf(1),djccf(1),djmonf(1),djmanf(1),djstart(1),djstop(1)

WARNINGS

Option -A selects process records from the accounting file /usr/adm/pacct by inference, since process genealogy is not available. Background processes having the same user-id and execution time window will be spuriously included.

BUGS

None

NAME

djsf - Distributed Job Status Facility

SYNOPSIS

djsf [-j job-id]
[-g group]
[-i file-id]
[-f file]
[-p pool]
[-c class]

DESCRIPTION

The Distributed Job Status Facility (djsf) command is a user facility, that can be utilized on any of the DJMS participating processors, to obtain status of background jobs that have been submitted into DJMS. This facility executes as a process and communicates with the Distributed Job Monitor (djmonf) process to obtain information on user jobs that may be waiting in a job pool or queue for execution or in execution at participating processors.

The parameters to djsf are:

-j job-id

The job-id returned by either djrf or djmonf upon a successful completion of a distributed job request. The job-id is comprised of a class, a hyphen (-), name, a period (.), and a unique number. If the job-id supplied is valid and the invoking user has permission, the distributed job request status will be returned.

-g group

The job group returned by either djrf or djmonf upon a successful completion of a distributed job request. If the job group supplied is valid and the invoking user has permission, the distributed job request(s) status will be returned.

-i file-id

The file-id returned by either djrf or djmonf upon a successful completion of a distributed job request. The file-id is comprised of a class, priority, a period (.), and a file system inode. If the request file-id supplied is valid and the invoking user has permission, the distributed job request status will be returned.

-f file

The request file which contains the actual distributed job request. If the request file is valid and the invoking user has permission, the distribution job request status will be returned.

-p pool

A complete path to the specified distributed job pool. If the supplied pool is valid and the invoking user has permission, for each distributed job request within the specified pool a status will be returned. The current class and pool configuration can always be obtained by viewing the file assigned to #define CFG in include file djms.h.

-c class

A class. If the supplied class is valid and the invoking user has permission, for each distributed monitor pool assigned to that class, for each distributed job request within the specified pool a status will be returned. The current class and pool configuration can always be obtained by viewing the file assigned to #define CFG in include file djms.h.

EXAMPLES

Status distributed job request job-id D-job0101.12345.

djsf -j D-job0101.12345

02/01/93 10:19:20 djsf : D-job0101.12345 DAILY 100D.1031 4 120.12 waiting

Status distributed job request job-id D-job0303.88551.

djsf -j D-job0303

02/01/93 10:19:20 djsf : D-job0303.88551 DAILY 100D.8741 1 8.25 executing

Status distributed job request group DAILY.

djsf -g DAILY

02/01/93 10:19:20 djsf : D-job0101.12345 DAILY 100D.1031 4 120.12 waiting
02/01/93 10:19:22 djsf : D-usr.18348 DAILY 100D.2045 2 136.44 waiting

Status distributed job request file-id 100D.1031.

djsf -i 100D.1031

02/01/93 10:19:20 djsf : D-job0101.12345 DAILY 100D.1031 4 120.12 waiting

Status distributed job request file /usr/djms/pool/0/1000.

djsf -f /usr/djms/pool/0/1000

02/01/93 10:19:20 djsf : D-usr.18348 D 100D.2045 0 136.44 waiting

Status distributed job pool /usr/djms/pool/0.

djsf -p /usr/djms/pool/0

02/01/93 10:19:22 djsf : D-usr.18348 D 100D.2045 0 136.44 waiting

Status distributed job execution class D.

djsf -c D

02/01/93 10:19:22 djsf : D-usr.18213 D 100D.2167 1 25.66 executing
02/01/93 10:19:24 djsf : D-job0103.22451 DAILY 130D.2046 2 45.65 waiting
02/01/93 10:19:25 djsf : D-usr.18233 D 101D.1004 3 30.73 waiting
02/01/93 10:19:29 djsf : D-job0203.31002 DAILY 100D.6651 4 15.23 waiting

WARNINGS

The -j,-g,-i,-f,-p, and -c option can never be supplied together. Also, complete file paths are required for both the -f and -p options. Option -i file-id is by far the quickest and most efficient way to status a job. Unfortunately, any job that utilizes the (djrf) -Q retry features may cause their file-id to change. For these jobs the -j job-id option may be more manageable.

FILES

/usr/djms/adm/cfg
/usr/djms/adm/log
/usr/djms/adm/lock
/usr/djms/adm/sar/*/*
/usr/djms/pool/*/*

SEE ALSO

djrf(1),djcf(1),djccf(1),djmonf(1),djmanf(1),djstart(1),djstop(1)

BUGS

None

NAME

djcf - Distributed Job Cancel Facility

SYNOPSIS

djcf [-j job-id]
[-g group]
[-i file-id]
[-f file]
[-p pool]
[-c class]

DESCRIPTION

The Distributed Job Cancel Facility (djcf) command is a user facility, that can be utilized on any of the DJMS participating processors, to cancel background jobs that have been submitted into DJMS. This facility executes as a process and communicates with the Distributed Job Monitor Facility(djmonf) slave daemon processes to cancel user jobs that may be waiting in a job pool for execution or in execution at participating processors. The djmonf processes carry-out the actual cancellations.

The parameters to djcf are:

-j job-id

The job-id returned by either djrf or djmonf upon a successful completion of a distributed job request. The job-id is comprised of a class, a hyphen (-), name, a period (.), and a unique number. If the job-id supplied is valid and the invoking user has permission, the distributed job request will be canceled.

-g group

The job group returned by either djrf or djmonf upon a successful completion of a distributed job request. If the job group supplied is valid and the invoking user has permission, the distributed job request(s) will be canceled.

-i file-id

The file-id returned by either djrf or djmonf upon a successful completion of a distributed job request. The file-id is comprised of a class, priority, a period (.), and a file system inode. If the request file-id supplied is valid and the invoking user has permission, the distributed job request will be canceled.

-f file

The request file which contains the actual distributed job request. If the request file is valid and the invoking user has permission, the distributed job request will be canceled.

-p pool

A complete path to the specified distributed job monitor pool. If the supplied pool is valid and the invoking user has permission, each distributed job request within the specified pool will be canceled. The current class and pool configuration can always be obtained by viewing the file assigned to #define CFG in include file djms.h.

-c class

A class. If the supplied class is valid and the invoking user has permission, each distributed monitor pool assigned to that class will be canceled. (i.e. Each distributed job request within the assigned pools will be canceled.) The current class and pool configuration can always be obtained by viewing the file assigned to #define CFG in include file djms.h.

EXAMPLES

Cancel distributed job request job-id D-job0101.12345.

djcf -j D-job0101.12345

02/01/93 10:19:20 djcf : D-job0101.12345 DAILY 100D.1031 canceled

Cancel distributed job request job-id D-job0303.88551.

djcf -j D-job0303

02/01/93 10:19:20 djcf : D-job0303.88551 DAILY 100D.8741 canceled

Cancel distributed job request group DAILY.

djcf -g DAILY

02/01/93 10:19:20 djcf : D-job0101.12345 DAILY 100D.1031 canceled
02/01/93 10:19:22 djcf : D-usr.18348 DAILY 100D.2045 canceled

Cancel distributed job request file-id 100D.1031.

djcf -i 100D.1031

02/01/93 10:19:20 djcf : D-job0101.12345 DAILY 100D.1031 canceled

Cancel distributed job request file /usr/djms/pool/0/1000.

djcf -f /usr/djms/pool/0/1000

02/01/93 10:19:20 djcf : D-usr.18348 D 100D.2045 canceled

Cancel distributed job pool /usr/djms/pool/0.

djcf -p /usr/djms/pool/0

02/01/93 10:19:22 djcf : D-usr.18348 D 100D.2045 canceled

Cancel distributed job execution class D.

djcf -c D

02/01/93 10:19:22 djcf : D-usr.18213 D 100D.2167 canceled
02/01/93 10:19:24 djcf : D-job0103.22451 DAILY 130D.2046 canceled
02/01/93 10:19:25 djcf : D-usr.18233 D 101D.1004 canceled
02/01/93 10:19:29 djcf : D-job0203.31002 DAILY 100D.6651 canceled

FILES

/usr/djms/adm/cfg
/usr/djms/adm/log
/usr/djms/adm/lock
/usr/djms/adm/sar/*/*
/usr/djms/pool/*/*

SEE ALSO

djrf(1),djsf(1),djccf(1),djmonf(1),djmanf(1),djstart(1),djstop(1)

WARNINGS

The -j,-g,-i,-f,-p, and -c option can never be supplied together. Also, complete file paths are required for both the -f and -p options. Option -i file-id is by far the quickest and most efficient way of canceling a job. Unfortunately, any job that utilizes the (djrf) -Q retry features may cause their file-id to change. For these jobs the -j job-id option may be more manageable.

BUGS

None

NAME

djccf - Distributed Job Configuration Change Facility

SYNOPSIS

djccf [-f cfg definitions]
[-p pool]
[-i]
[-y]

DESCRIPTION

The Distributed Job Configuration Change Facility (djccf) command is an administrative change facility used to dynamically configure how many networked processors and concurrent Distributed Job Monitor Facility (djmonf) processes will participate in DJMS. This facility executes as a process and communicates with the master daemon process (the Distributed Job Manager Facility (djmanf)) via UDP/IP datagrams over the network to achieve its main objective. Basically, it provides the administrator with the ability to tune and adapt the load sharing and load balancing distributed system to better meet user processing needs. It also provides a processing throttle or governor mechanism in that the number of concurrent user background executions across the participating processors can be controlled efficiently and automatically. Configuration changes can be performed either interactively or in the background.

The parameters to djccf are:

-f cfg definitions

The cfg definitions file that contains the new layout/configuration for all participating processors. This file, if valid, will replace file #define CFG in include file djms.h. If this option is not supplied, the current configuration will be utilized.

-p pool

The UNIX file directory that contains all defined distributed job monitor NFS pools. If this option is not supplied, #define POOL in include file djms.h will be utilized.

-i

Interactively utilize the vi(1) command to modify either the current or supplied layout/configuration. Once verified, the modifications made will replace file #define CFG in include file djms.h. Automatically upon verification failure, djccf requires the modified configuration to be corrected. The visual editor vi(1) must be installed for this option to be supported.

-y

Automatically respond "YES" to any questions asked. This option is primarily for background oriented executions.

EXAMPLES

Change the current DJMS configuration to what is defined in file /usr/djms/adm/cfg.

```
cat /usr/djms/adm/cfg

#
# DJMS Configuration File
#
# Master
#
#M    C     S     Host    Port   Description
#
 0    N     A     lion    6000   ai SPARCstation 1
#
# Slaves
#
#M    C     S     Host    Port   Description
#
 1    A     A     lion    0      ai SPARCstation 1 SunOS 4.1.3
 2    A     A     lion    0      ai SPARCstation 1 SunOS 4.1.3
 3    A     A     lion    0      ai SPARCstation 1 SunOS 4.1.3
 4    A     A     lion    0      ai SPARCstation 1 SunOS 4.1.3
 5    B     A     lion    0      ai SPARCstation 1 SunOS 4.1.3
 6    B     A     pluto   0      ai SPARCstation SLC SunOS 4.1.3
 7    B     A     pluto   0      ai SPARCstation 1 SunOS 4.1.3
 8    C     I     lion    0      ai SPARCstation 1 SunOS 4.1.3
 9    T     A     jupiter 0      ai Sun 3/80 SunOS 4.1.3
10    T     A     jupiter 0      ai Sun 3/80 SunOS 4.1.3
11    T     I     jupiter 0      ai Sun 3/80 SunOS 4.1.3

djccf -f /usr/djms/adm/cfg -y

11/19/93 15:24:40 djccf  : install new configuration? y
11/19/93 15:24:44 djccf  : A 4 active 0 disabled 0 inactive configured
11/19/93 15:24:44 djccf  : B 3 active 0 disabled 0 inactive configured
11/19/93 15:24:44 djccf  : C 0 active 0 disabled 1 inactive configured
11/19/93 15:24:44 djccf  : T 2 active 0 disabled 1 inactive configured
```

```
11/19/93 15:24:44 djccf  : 0 of 12 daemons executing
11/19/93 15:24:44 djccf  : activate daemons to initialize configuration? y
11/19/93 15:24:47 djccf  : daemons have been activated
```

FILES

/usr/djms/adm/cfg
/usr/djms/adm/log
/usr/djms/adm/lock
/usr/djms/adm/sar/*/*
/usr/djms/pool/*/*

SEE ALSO

djrf(1),djsf(1),djcf(1),djmonf(1),djmanf(1),djstart(1),djstop(1)

WARNINGS

Complete file paths are required for all -f and -p options.

BUGS

None

NAME

djmonf - Distributed Job Monitor Facility

SYNOPSIS

djmonf [-p monitor pool]
[-L log output file]

DESCRIPTION

The Distributed Job Monitor Facility (djmonf) daemon, the slave process, is a system facility to execute background jobs on a participating processors on the network. This facility may execute as a connection less server process on each processor defined in the configuration. The sole purpose of djmonf is to run the receiver initiated portion of the DJMS algorithm and also to execute, monitor, terminate, and report information about user background jobs. It will also ensure automatic re-start of user background jobs after processor or network failures. This slave facility communicates, synchronizes, and cooperates with its one and only master Distributed Job Management Facility (djmanf) process at the control processor. The local scheduling policy is strictly managed by the djmonf process at each processor. The djmonf process services jobs or tasks in a first-in-first-out (fifo) manner where both priority of the job and elapsed wait time in-queue have bearing. On demand by the Distributed Job Management Facility (djmanf), djmonf will be signaled or remotely spawned on a participating processor to schedule a particular job or task request. Once djmonf has started, each job request contained in its assigned input job pool is processed. Each job or task is executed and monitored. If the job or task fails, subsequent executions may take place, but not necessarily on the same processor. During each job execution and after each successful job completion or failure, pertinent information is transferred over the network via UDP/IP data grams to djmanf daemon in order to update the global DJMS state and possibly triggers subsequent job migrations. After all jobs have been completed and the designated input job pool is empty, djmonf will message djmanf of its termination and will terminate. On demand, djmonf can be signaled to quiesce or cancel a distributed job. The djmonf process (if executing) will signal the current job command *UNIX* process to terminate. Once the job or task has terminated, djmonf will message djmanf of its termination. During a quiesce, djmonf will shutdown immediately and terminate.

The parameters to djmonf are:

-p monitor pool

The *UNIX* RFS or NFS directory that contains all distributed job requests for a particular processor.

-L log output file

A *UNIX* RFS or NFS file where all stdout/stderr print messages will be written. If this option is not supplied, log output file will default to LOG.

EXAMPLES

Run the Distributed Job Monitor Facility in the background utilizing *UNIX* NFS directory /usr/djms/pool/1 as the defined monitor pool and UNIX NFS file /usr/djms/adm/log as the defined log output file.

```
djmonf -p /usr/djms/pool/1 -L /usr/djms/adm/log &
```

FILES

/usr/djms/adm/cfg
/usr/djms/adm/log
/usr/djms/adm/lock
/usr/djms/adm/sar/*/*
/usr/djms/pool/*/*

SEE ALSO

djrf(1),djsf(1),djcf(1),djccf(1),djmanf(1),djstart(1),djstop(1)

WARNINGS

A complete file path is required for the -i option. Also, if the -L option is supplied, the same complete file path rule applies.

BUGS

None

NAME

djmanf - Distributed Job Management Facility

SYNOPSIS

djmanf [-p pool]
[-P polling interval]
[-L log output file]

DESCRIPTION

The Distributed Job Management Facility (djmanf) is a system facility, a connection less server which acts as the master daemon process at one of the networked processors; the control processor on the network. Its sole purpose is to perform the sender initiated and adaptive portions of the DJMS algorithm. In doing so, it executes, monitors, and terminates its slaves, the Distributed Job Monitor Facility (djmonf) processes, while maintaining the global state of DJMS. By gathering information from each of its slave processes executing on each of the participating processors, djmanf records and maintains the global state in its control processor's shared memory segment as well as on dedicated networked disk storage. Based on this global information, the Distributed Job Manager process deploys its load sharing policies. It will also ensure automatic re-start of failed slave processes after processor or network failures. It supports a single point of control for DJMS initiation and termination. This master facility will communicate and synchronize itself with each of its djmonf slaves. The Distributed Job Manager process manages a transfer, selection, location, information gathering, and stability control set of policies. The transfer policy is based solely on the comparison of each participating processor's CPU queue length to the average CPU queue length of all participating processors within a designated class. Very simply, jobs are migrated between job pools within a class to share and balance the processing load. During a migration, queued jobs may shift from one processor to an other until executed. The selection policy is based solely on identifying jobs or tasks that are not presently executing on a processor. Again, DJMS only supports non-preemptive transfers. The location policy is based solely on the pre-defined DJMS configuration as well as CPU queue lengths. This configuration dictates for a given job class how many jobs can execute concurrently as well as on which participating processors on the network. The information gathering policy is managed by two methods. First, a message passing model exists where the master process at the control processor communicates via UDP/IP data grams over the network to all its slave processes at participating processors. Second, the master process on a event trigger basis polls each slave's job pool for CPU queue length information. Through these information gathering techniques, the master process has accurate global knowledge as to what is occurring in the distributed system. Stability control is ensured cooperatively by the master djmanf process and the Distributed Job Request Facility (djrf) process.

As the distributed system load becomes increasingly heavy, the master djmanf process and the djrf processes ensure that for each class all participating slave processors have evenly dispersed CPU queue lengths. As long as the jobs or tasks are evenly dispersed among processors, the sender initiated transfer policy becomes dormant. In general, DJMS minimizes the inefficiences of endless sender initiated transfers under heavy system loads. At heavy system loads, transfer or migration are solely triggered by a djmonf running the receiver initiated portion of the DJMS algorithm as jobs or tasks complete and depart. On demand by the Distributed Job Request Facility (djrf) process or periodically, djmanf will awaken via a receipt of UDP/IP message to schedule Distributed Job Monitor Facility (djmonf) processes. Once djmanf has awakened, all defined configuration job pools are polled through NFS or RFS for the presence of job requests. Each configured djmonf process whose job pool contains job requests will be executed (if not already executing). After all necessary configuration job monitor scheduling has been completed successfully, djmanf will listen for subsequent events. On demand by receipt of a djmonf UDP/IP data gram, djmanf will awaken to recognize that the particular Distributed Job Monitor Facility (djmonf) has started or terminated. On demand by a specific djmonf process, djmanf will awaken to gather global DJMS state information. Once djmanf has awakened and recognized that a Distributed Job Monitor has terminated or a job request has completed, djmanf will again poll through NFS or RFS the job pools for the presence of distributed job requests. Again distributed monitor scheduling may take place after all load balancing via job migration has completed. The djmanf process will again listen for subsequent events. On demand, djmanf could be awakened to quiesce all executing Distributed Job Monitor Facilities. Once djmanf has awakened and recognized the quiesce, djmanf will immediately ignore any incoming requests from djrf processes and will signal via a *UNIX* kill system call each running Distributed Job Monitor (djmonf) process to terminate. Once all Distributed Job Monitor (djmonf) processes have reported their termination, djmanf will terminate as well.

The parameters to djmanf are:

-p pool

The *UNIX* RFS or NFS directory that contains all defined configured monitor disk pools.

-P polling interval

The number of seconds between polls for a presence check of distributed job requests and possible load balancing via job migrations.

-L log output file

A *UNIX* RFS or NFS file where all stdout/stderr print messages will be written. If this option is not supplied, log output file will default to LOG.

EXAMPLES

Run the Distributed Job Management Facility in the background polling every 5 minutes utilizing *UNIX* NFS directory /usr/djms/pool as the defined disk pool and *UNIX* NFS file /usr/djms/adm/log as the defined log out file. All messages from stdout and stderr will be written to log output file /usr/djms/adm/log.

```
djmanf -p /usr/djms/pool -P 300 -L /usr/djms/adm/log &
```

FILES

/usr/djms/adm/cfg
/usr/djms/adm/log
/usr/djms/adm/lock
/usr/djms/adm/sar/*/*
/usr/djms/pool/*/*

SEE ALSO

djrf(1),djsf(1),djcf(1),djccf(1),djmonf(1),djstart(1),djstop(1)

WARNINGS

A complete file path is required for the -p option. Also, if the -L option is supplied, the same complete file path rule applies.

BUGS

None

NAME

djstart - Distributed Job Start

SYNOPSIS

djstart

DESCRIPTION

The Distributed Job Start (djstart) command is system administrative facility used to start up the Distributed Job Management System (DJMS). The djstart command can be executed on any of the participating processors as defined in the DJMS configuration. Typically, this facility is executed on the DJMS master control processor. The djstart facility starts the Distributed Job Management Facility (djmanf) on the master processor to execute and monitor all load sharing and load balancing activities.

EXAMPLES

Start the Distributed Job Management System from the *UNIX* prompt in the background.

```
djstart &

Distributed Job Management System (DJMS) Start-up
Version 2.0 1992, 1993 all rights reserved


Distributed Job Management System (DJMS) Started
All facilities are up and available
```

FILES

/usr/djms/adm/cfg
/usr/djms/adm/log
/usr/djms/adm/lock
/usr/djms/adm/sar/*/*
/usr/djms/pool/*/*

SEE ALSO

djrf(1),djsf(1),djcf(1),djccf(1),djmonf(1),djmanf(1),djstop(1)

WARNINGS

None

BUGS

None

NAME

djstop - Distributed Job Stop

SYNOPSIS

djstop

DESCRIPTION

The Distributed Job Stop (djstop) command is system administrative facility used to shut-down the Distributed Job Management System (DJMS). The djstop command can be executed on any of the participating processors as defined in the DJMS configuration. Typically, this facility is executed on the DJMS master control processor. The djstop facility initiates the DJMS quiesce via delivery of a datagram over the network to the Distributed Job Management Facility (djmanf) daemon process. Once djmanf receives the shut-down message, it carries out the complete quiesce process.

EXAMPLES

Stop the Distributed Job Management System from the *UNIX* prompt.

```
djstop

Distributed Job Management System (DJMS) Shut-down
Version 2.0 1992, 1993 all rights reserved


Distributed Job Management System (DJMS) Stopped
All facilities are down and unavailable
```

FILES

/usr/djms/adm/cfg
/usr/djms/adm/log
/usr/djms/adm/lock
/usr/djms/adm/sar/*/*
/usr/djms/pool/*/*

SEE ALSO

djrf(1),djsf(1),djcf(1),djccf(1),djmonf(1),djmanf(1),djstart(1)

WARNINGS

None

BUGS

None

Vita

Michael Kenneth Nemeth was born in Phillipsburg, New Jersey, U.S.A on August 29, 1965, to John Lewis Nemeth and Marlene Ann Piatt. Mr. Nemeth earned his Bachelor of Science degree in Information and Computer Science from the University of Pittsburgh in May of 1987. In June of 1987, Mr. Nemeth began his professional career as a System Analyst and Programmer with NCR in Dayton, Ohio. In July of 1988, Mr. Nemeth continued in his career as a Systems Designer with AT&T Microelectronics in Allentown, Pennsylvania. In November of 1989, Mr. Nemeth advanced in his career as a Technical Staff Member and transferred to AT&T Microelectronics in Berkeley Heights, New Jersey. In July of 1993, Mr. Nemeth continued his career advancement and is presently a Senior Technical Staff Member. During Mr. Nemeth's professional career, his work has focused on Operating Systems, Data Networking, and Distributed Systems.

END

OF

TITLE