DOKUZ EYLÜL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

# DISTRIBUTED COMPUTING ON BEOWULF CLUSTERS

by

Oğuz AKAY

February, 2008

İZMİR

# DISTRIBUTED COMPUTING ON BEOWULF CLUSTERS

**A Thesis Submitted to the**
**Graduate School of Natural and Applied Sciences of Dokuz Eylül University**
**In Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Computer Engineering, Computer Engineering Program**

**by**
**Oğuz AKAY**

**February, 2008**
**İZMİR**

**Ph.D. THESIS EXAMINATION RESULT FORM**

We have read the thesis entitled **“DISTRIBUTED COMPUTING ON BEOWULF CLUSTERS”** completed by **OĞUZ AKAY** under supervision of **PROF. DR. ALP KUT** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

Supervisor

| | |
|---|---|
| Thesis Committee Member | Thesis Committee Member |
| Examining Committee Member | Examining Committee Member |

Prof.Dr. Cahit HELVACI
Director
Graduate School of Natural and Applied Sciences

**ACKNOWLEDGEMENTS**

I would like to express my sincere gratitude to my advisor, Prof. Dr. Alp Kut, for his confidence in me, and for his guidance.

I am grateful to Assoc. Prof. Dr. Yalçın Çebi and Asst. Prof. Dr. Ahmet Özkurt for spending their time for this thesis in term examinations and for their valuable advises.

I am indepted to my brother, Asst. Prof. Dr. Olcay Akay for his help.

I would like to thank my managers in Astron, Yaşar Holding for supporting my academic career.

Finally, my special thanks go to my parents and to my wife, Filiz, for their love, support, and patience during the last six years.

Oğuz AKAY

## DISTRIBUTED COMPUTING ON BEOWULF CLUSTERS

## ABSTRACT

Building low cost Beowulf style clusters by using tens or hundreds of PCs is a popular method to achieve higher computing capacities. To gain advantages of such a computing platform, a load balancing scheme is needed for transparent distribution of loads of individual computers throughout the whole cluster in a scalable and efficient manner.

In this thesis a scalable cluster architecture and a load balancing model for heterogeneous Beowulf cluster environments are presented. For scalability issues the system relies on a hierarchically centralized architecture. To have general purpose characteristics the proposed load balancing model offers some dynamic and customizable properties in its design. For this purpose, multiple user defined load indices are considered in load calculations like CPU utilization, memory usage, network bandwidth capacity, etc. along with their combinations. In addition, the load distribution policy is based on customizable adaptive load threshold values that dynamically adjust the load distribution decisions according to the system state.

The thesis details the design, implementation and performance evaluations of proposed models.

**Keywords** : beowulf, cluster, load balancing, heterogeneous, adaptive, dynamic, hierarchical

**BEOWULF KÜMELERİ ÜZERİNDE**
**DAĞITIK İŞLEM YÜRÜTME**

## ÖZ

Daha yüksek işlem kapasitesi elde etmek amacıyla onlarca, hatta yüzlerce kişisel bilgisayarı (PC) kullanarak düşük maliyetli Beowulf tipi kümeleri oluşturmak son yıllarda eğilim kazanmıştır. Bu tür bir işlem platformunun avantajlarından faydalanabilmek için işi kümenin içindeki bilgisayarlar arasında etkin bir biçimde ve saydam olarak paylaştıracak, ölçeklenebilir özellikte bir yük dengeleme modeli gerekmektedir.

Bu tezde, heterojen yapıdaki Beowulf kümeleri için tasarlanan ölçeklenebilir bir küme mimarisi ile bu mimari üzerine inşa edilmiş bir yük degeleme modeli sunulmaktadır. Ölçeklenebilir bir yapı için sistem hiyerarşik olarak merkezileştirilmiş bir mimari üzerine yapılandırılmıştır. Genel amaçlı kullanım özelliği kazandırmak amacıyla, önerilen yük dengeleme modeli, tasarımında bazı dinamik ve uyarlanabilir unsurlar barındırmaktadır. Bu amaçla, yük değerleri hesaplamasında, CPU durumu, bellek kullanımı, ağ arabirim bantgenişliği gibi farklı kombinasyonlarda birden çok yük endeksi hesaplamaya dahil edilebilmektedir. Buna ek olarak yük dağıtım modeli işleyişi, değiştirilebilir tarzda tasarlanmış ve sistemin anlık durumuna göre yük dağıtma kararlarını dinamik olarak değiştirebilen uyarlanabilir yük eşik değerlerine dayandırılmıştır.

Tez içerisinde, önerilen modellerin tasarım, uygulama ve performans değerlendirmeleri detayları ile yer almaktadır.

**Anahtar sözcükler**: beowulf, küme, yük dengeleme, heterojen, uyarlanabilir, dinamik, hiyerarşik

**CONTENTS**

# CHAPTER ONE
# INTRODUCTION

## 1.1 Area of Research

Cluster computing is a technology of connecting multiple computers together to behave like a single computer. Clustering is generally used for high performance parallel processing, load balancing and high availability.

Clustering is almost as old as mainframe computing. From the earliest days, developers wanted to create applications that needed more computing power than a single system could provide. Then applications that could take advantage of computing in parallel were develeoped to run on multiple processors at once. Clusters can also enhance the reliability of a system, so that failure of any one part would not cause the whole system to become unavailable.

After the mainframes, mini-computers and technical workstations were also connected in clusters. These systems used special hardware and special interconnect hardware and communications protocols. The challenge with these special clusters is that the hardware and software tend to be very expensive, and vendors may stop support of the product. Some vendors proposed open clusters built on their operating systems and commodity hardware. Although neither of these proposed clustering environments were deployed, the idea of using off-the-shelf hardware to build clusters was underway. Now many of the largest clusters in existence are based on standard PC hardware, often running Linux.

Networks of Workstations (NOW) technology has been introduced to be a viable replacement for conventional supercomputers performing trillions of calculations per second (Castanegra, Cheng, & Fatoohi, 1994). Linux based Beowulf clusters built on that concept were used in some areas requiring high performance computing (HPC) where parallel computers run some specialized applications allowing scientific institutions and enterprises to perform computations, modeling, rendering, simulations, visualizations and other sorts of tasks that a few years ago were limited to very large computer centers (Sterling, Becker, Dorband, Savarese, Ranawake, & Packer, 1995). They were widely used than any other type of parallel computer because of their low cost, flexibility, and accessibility (Dongarra, Sterling, Simon, & Strohmaier, 2005). While usually clusters are constructed by tightly-connected computers, they are also adapted to utilize the idle time of nondedicated loosely coupled workstations (Soria, Pérez-Segarra, & Oliva, 2002).

A Beowulf cluster is a distributed system consisting of inexpensive computers, built from off-the-shelf components, connected together cheaply, usually in an Ethernet network infrastructure (Meredith, Carrigan, Brockman, Cloninger, Privoznik, & Williams, 2003). It enables leveraging the investment already made in PCs and workstations. In addition, it is relatively easy to increase the computing capacity by simply adding new PCs to the network.

The use of clusters for the purpose of load balancing is a very popular subject. Due to their massively parallel nature, clusters were generally used to solve complex computational problems in many areas like 3D modelling, neural networks and biology. These clusters require specialized parallel applications designed for the problem, like those written using the MPI (Walker, & Dongarra, 1996) or PVM (Sunderam, Geist, Dongarra, & Manchek, 1994) libraries. The factor that differentiates load balancing approach from the others is the lack of a single parallel program that runs on each node of the cluster. Instead, there is a load balancing component, usually called the distributed task scheduler, that usually runs a specific algorithm to distribute the workload across the nodes of the cluster. Ideally, the load distribution scheme tries to balance the workload among the machines in the cluster, decreasing response times and increasing overall throughput (Shivaratri, Krueger, & Singhal, 1992).

The explicit software design requirement of parallel applications running on Beowulf clusters is sometimes a problem. Since it is the job of the software to distribute the work by dividing it to subproblems to be executed in a parallel manner on different nodes of the cluster, a program written in nondistributed style has to be converted into a distributed application with an embedded load distribution mechanism to run on a Beowulf cluster (Adams, 2005). The concept of load balancing in clusters was suggested to solve this issue and hide the load distribution details from the application and its programmer. In this concept, instead of a parallel application, usually a middleware component or subsystem, the distributed scheduler, as in PANTS (Claypool, & Finkel, 2002) and CONDOR (Thain, Tannenbaum, & Livny, 2005) isolates the distributed resources from applications and transparently distributes the workload throughout the cluster.

In order for a cluster to be scalable, it must ensure that each server is fully utilized. The standard technique for accomplishing this is load balancing. The basic idea behind load balancing is that by distributing the load proportionally among all the servers in the cluster, the servers can each run at full capacity, while all requests receive the lowest possible response time. In a web server scenario, load-balancing refers to the technique of routing user requests over a certain number of networked computers, so as to keep the average usage of any system's resource approximately the same within that network that acts as a functional unit.

In distributed systems, an attempt to distribute workload equally involves very high computational overheads. Most of the work is spent on collecting global state. If the applications considered demonstrates a pattern of frequent communication and synchronization, this global state changes rapidly, making load balancing unviable. Furthermore, if the grain size of the transfered work is not big enough to amortize the load balancing overheads, load balancing is not preferable even if the balancer algorithm guarantees accurate decisions based on global system state. In these situations, an alternative to distributing workload equally, is to ensure that all nodes are busy. Reducing idle times, and thereby the total program execution time is a far more preferable objective than attempting to distribute the workload equally. This strategy is called load sharing. Most systems implement load sharing rather than load balancing. These two terms are now being used interchangeably.

In general, clusters can provide both high availability and scalability for important computer applications such as business, medical and scientific applications. Much research has gone in to clustering technology over recent years, and quite a few solutions exist to provide load balancing services.

## 1.2 Scope of Research

The research presented in this thesis primarily addresses the problem of building scalable cluster architectures and utilizing the capacity of resources of such systems by efficient distribution of loads through these resources. The cluster components are ordinary independent personal computers with similar architectures but different hardware specifications (CPU, memory, disk, network interface, etc.) running Linux operating system. These computers are connected via a high speed (such as ethernet) local or campus area network and they can communicate with each other directly by a common network and transport protocol supporting unicast and multicast communication methods (like UDP/IP).

The research focuses on the arrangement of the computing resources and load distribution techniques on general purpose Beowulf clusters rather than those designed for running specialized applications to solve specific problems. For this reason, task types, patterns or their behaviours are not considered. Besides the organization of data storage (central or distributed file systems and directory structures, replication of data, etc.) is not in the scope of this research.

## 1.3 Research Objectives

The aim of the project is designing a load balancing model for Beowulf style cluster systems. The model to be designed targets some benefits to cluster systems. Some of these are:

- **Scalability:** Cluster systems are scalable in that performance can be increased beyond that of a single node by adding more nodes to the cluster.

This is a great advantage in that if the load that is needed to share expands beyond expectation, simply extra hardware is added to the cluster to increase its capacity. The system should handle tens to hundreds of nodes effectively with reasonable and foreseeable overhead.

- **Availability:** It is a measure of how well a computer system can continuously deliver services to clients. Because of the failover features of most modern cluster technology, it is much more likely that the cluster will be available to offer services to its clients, as it is unaffected by most failures in individual parts of the cluster. The other nodes will each have to deal with the small increase in traffic that they will experience because of the failure of one node, but the result is usually not catastrophic.

- **Manageability/Flexibility:** Cluster management systems offer software to inspect the overall status of the cluster, to perform manual load balancing and set parameters for automatic load balancing and to perform rolling upgrades of software.

- **Lower total cost of ownership:** Because of the reduction on downtime that cluster-based load balancing provides, the cost of administrative support for the system and also the amount of money that is lost through downtime is reduced.

There are some considerations for the design of the model. These are:

- **Heterogeneous resources:** Heterogeneous cluster systems are multiprocessor systems that may have nodes of dissimilar types. Design freedom can lead to heterogeneity as machines can have;

- Different processor speeds,
- Different memory sizes,
- Different I/O speeds,

- Different network interface speeds.

The heterogeneity may even include nodes that have processors of dissimilar architectures, which have distinct instruction sets, byte orderings, and different operating systems, but this type of heterogeneity is not scope of this research.

- **Dynamic loads:** the system has a dynamic nature, that is the load balancing scheme makes the load distribution decisions at runtime, without any prior knowledge about the load patterns (task types, their submission times, nodes to be submitted, etc.).

As to be discussed in the next chapter, dynamic load distribution algorithms use system state information (the loads at nodes), at least in part, to make the load distribution decisions, which static algorithms make no use of such information.

- **Adaptivity of operations:** The model has adaptive characteristics, meaning that the load distribution scheme adjusts its activities with respect to the current state of the system. Moreover the level of adaptivity can be adjusted by customizing some parameters of the system according to needs.

Adaptive load distribution systems adapt their activities by dynamically changing the parameters of the algorithm to suit changing of the system state. For example, an adaptive system may adjust its activities at high system load states to prevent imposing extra overhead.

- **Load indices:** The system considers combination of multiple load indices to calculate the load levels of the nodes. Moreover the selection of the load indices is a customizable process. Hence, it is configurable according to needs that more than one load index can be selected to be involved load calculations with different importance factors (weights).

Load is the demand or usage of some system resource. The load metric is used to determine if a node is “free" or “busy". In other words, the load metric is used to decide if the machine should attempt to lessen it's load by transferring tasks, or take on more load by accepting tasks from other machines in the cluster.

A load index of a node can be comprised of a number of things;

- CPU queue length,
- CPU usage,
- Idle process run time,
- CPU load average,
- Average response time,
- Memory usage,
- Memory page-fault rate,
- I/O queue length,
- I/O service time,
- I/O blocks read/written,
- Network bandwidth utilization,
- Context switching,
- Interrupts,
- Task arrival rate,
- etc.

## 1.4 Outline of the Thesis

Chapter 2 discusses some theoretical and background information around the area of research. In Chapter 3 the design of hierarchically layered the cluster architecture and its fault tolerant management model is described. Chapter 4 presents the load balancing model in detail. The experiments and their results about measuring the performance of the models are discussed in Chapter 5. Finally, Chapter 6 contains some concluding remarks and suggestions for future work.

# CHAPTER TWO
# RELATED WORK

## 2.1 Cluster Computing

### *2.1.1 History of Clustering*

The computing industry is one of the fastest growing industries and it is fueled by the rapid technological developments in the areas of computer hardware and software. The technological advances in hardware include chip development and fabrication technologies, fast and cheap microprocessors, as well as high bandwidth and low latency interconnection networks. Software technology is also developing fast. Operating Systems, programming languages, development methodologies, and tools, are now available. This has enabled the development and deployment of applications catering to scientific, engineering, and commercial needs (Baker, & Buyya, 1999).

From the earliest days, developers wanted to create applications that needed more computing power than a single system could provide. Then came applications that could take advantage of computing in parallel, to run on multiple processors at once (Harbaugh, 2004). The main reason for creating and using parallel computers is that parallelism is one of the best ways to overcome the speed bottleneck of a single processor. In addition, the price performance ratio of a small cluster-based parallel computer as opposed to a minicomputer is much smaller and consequently a better value. In short, developing and producing systems of moderate speed using parallel architectures is much cheaper than the equivalent performance of a sequential system. In addition, clusters can enhance the reliability of a system, so that failure of any one part would not cause the whole system to become unavailable.

The taxonomy of cluster systems is based on how their processors, memory, and interconnect are laid out. The most common systems are (Baker, & Buyya, 1999):

- Massively Parallel Processors (MPP)

- Symmetric Multiprocessors (SMP)
- Cache-Coherent Nonuniform Memory Access (CC-NUMA)
- Distributed Systems
- Clusters

Table 2.1 shows a modified version comparing the architectural and functional characteristics of these machines (Hwang, & Xu, 1998).

Table 2.1 Key characteristics of scalable parallel computers

| **Charac-teristic** | **MPP** | **SMP CC-NUMA** | **Cluster** | **Distributed** |
|---|---|---|---|---|
| Number of Nodes | O(100)-O(1000) | O(10)-O(100) | O(100) or less | O(10)-O(1000) |
| Node Complexity | Fine grain or medium | Medium or coarse grained | Medium grain | Wide Range |
| Internode communi-cation | Message passing/ shared variables for distributed shared memory | Centralized and Distributed Shared Memory (DSM) | Message Passing | Shared files, RPC, Message Passing and IPC |
| Job Scheduling | Single run queue on host | Single run queue mostly | Multiple queue but coordinated | Independent queues |
| SSI Support | Partially | Always in SMP and some NUMA | Desired | No |
| Node OS copies and type | N micro-kernels monolithic or layered OSs | One monolithic SMP and many for NUMA | N OS platforms -homogeneous or micro-kernel | N OS platforms homogeneous |
| Address Space | Multiple - single for DSM | Single | Multiple or single | Multiple |
| Internode Security | Unnecessary | Unnecessary | Required if exposed | Required |
| Ownership | One organization | One organization | One or more organizations | Many organizations |

An MPP is usually a large parallel processing system with a shared-nothing architecture. It typically consists of several hundred processing elements (nodes), which are interconnected through a high-speed interconnection network/switch. Each node can have a variety of hardware components, but generally consists of a main memory and one or more processors. Special nodes can, in addition, have peripherals such as disks or a backup system connected. Each node runs a separate copy of the operating system.

SMP systems have from 2 to 64 processors and can be considered to have shared-everything architecture. In these systems, all processors share all the global resources available (bus, memory, I/O system); a single copy of the operating system runs on these systems.

CC-NUMA is a scalable multiprocessor system having a cache-coherent nonuniform memory access architecture. Like an SMP, every processor in a CC-NUMA system has a global view of all of the memory. This type of system gets its name (NUMA) from the nonuniform times to access the nearest and most remote parts of memory.

Distributed systems can be considered conventional networks of independent computers. They have multiple system images, as each node runs its own operating system, and the individual machines in a distributed system could be, for example, combinations of MPPs, SMPs, clusters, and individual computers.

At a basic level a cluster is a collection of workstations or PCs also called NOWs (Networks of Workstations) that are interconnected via some network technology (Baker, & Buyya, 1999). For parallel computing purposes, a cluster will generally consist of high performance workstations or PCs interconnected by a high-speed network. A cluster works as an integrated collection of resources and can have a single system image spanning all its nodes. Such a cluster can provide fast an reliable services to computationally intensive applications.

In the 1980s, it was believed that computer performance was best improved by creating faster and more efficient processors. This idea was challenged by parallel processing, which in essence means linking together two or more computers to jointly solve some problem. Since the early 1990's there has been an increasing trend to move away from expensive and specialised propriety parallel supercomputers towards networks of workstations. Among the driving forces that have enabled this transition has been the rapid improvement and availability of commodity high-performance components for workstations and networks. These technologies are making networks of computers (PCs or workstations) an appealing vehicle for parallel processing and this is consequently leading to low-cost commodity supercomputing (Baker, & Buyya, 1988).

Clusters built with off-the-shelf hardware are generally AMD or Intel-based servers, networked with gigabit Ethernet, and using Infiniband, MyriNet, SCI, or some other high-bandwidth, low-latency networks for the interconnect; the inter-node data transfer network. Linux is becoming the cluster OS of choice, due to its similarity to UNIX, the wide variety of open-source software already available, as well as the strong software development tools available.

The use of parallel processing as a means of providing high performance computational facilities for large-scale and grand-challenge applications has been investigated widely. Until recently, however, the benefits of this research were confined to the individuals who had access to such systems. The trend in parallel computing is to move away from specialized traditional supercomputing platforms, such as the Cray/SGI T3E, to cheaper, general purpose systems consisting of loosely coupled components built up from single or multiprocessor PCs or workstations. This approach has a number of advantages, including being able to build a platform for a given budget which is suitable for a large class of applications and workloads.

The use of clusters to prototype, debug, and run parallel applications is becoming an increasingly popular alternative to using specialized, typically expensive, parallel computing platforms. An important factor that has made the usage of clusters a practical proposition is the standardization of many of the tools and utilities used by parallel applications. Examples of these standards are the message passing library MPI and parallel virtual machine PVM. In this context, standardization enables applications to be developed, tested, and even run on NOW, and then at a later stage to be ported, with little modification, onto dedicated parallel platforms where CPU-time is accounted and charged.

The following list highlights some of the reasons NOW is preferred over specialized parallel computers (Baker, & Buyya, 1999) :

- Individual workstations are becoming increasingly powerful. That is, workstation performance has increased dramatically in the last few years and is doubling every 18 to 24 months. This is likely to continue for several years, with faster processors and more efficient multiprocessor machines coming into the market.

- The communications bandwidth between workstations is increasing and latency is decreasing as new networking technologies and protocols are implemented in a LAN.

- Workstation clusters are easier to integrate into existing networks than special parallel computers.

- Typical low user utilization of personal workstations.

- The development tools for workstations are more mature compared to the contrasting proprietary solutions for parallel computers, mainly due to the nonstandard nature of many parallel systems.

- Workstation clusters are a cheap and readily available alternative to specialized high performance computing platforms.

- Clusters can be easily grown; node's capability can be easily increased by adding memory or additional processors.

Clearly, the workstation environment is better suited to applications that are not communication-intensive since a LAN typically has high message start-up latencies and low bandwidths. If an application requires higher communication performance, the existing commonly deployed LAN architectures, such as Ethernet, are not capable of providing it.

### *2.1.2 Beowulf Clusters*

Beowulf is a project to produce parallel Linux clusters from off-the-shelf hardware and freely available software. Conceived in 1994 at the Goddard Space Flight Center, there are now dozens of Beowulf-class systems in use in Government and at Universities worldwide. Many of these organisations have joined to form a Beowulf consortium who actively share information and software for Beowulf systems. Some members include: Caltech, Los Alamos National Laboratory, Oak Ridge National Laboratory, Sandia National Laboratory, Duke, Oregon, Clemson and Washington Universities, The US National Institute of Health (NIH), as well as DESY in Germany, Kasetsart University in Thailand. NAS, Goddard Space Flight Center, Ames and various NASA sites and divisions have built major Beowulf systems. Other small systems have also been built at the University of Southern Queensland and the University of Adelaide amongst many other sites (Dickson, Homic, & Villamin, 2000) .

The original Beowulf parallel workstation prototyped by NASA combined sixteen 486DX PC's with dual Ethernet networks, 0.5 GByte of main memory, and 20 GBytes of storage, and providing up to eight times the disk I/O bandwidth of conventional workstations. Since the Beowulf design uses commodity hardware components and freely available systems software, NASA's project has demonstrated how the price/performance ratio of this route is attractive for many academic and research organisations.

One of the most difficult tasks in designing and commissioning a Beowulf cluster is tracking the cost/performance benefits from the multitude of different possible configuration options.

Broadly the design choices in order of importance for performance are:

1. Processor/Platform (eg PC, iMac, Alpha, O2,...)
2. Network infrastructure (Ethernet, Fast Ethernet, Myrinet, SCI,...)
3. Disk configuration (Diskless, EIDE or SCSI interface...)
4. Operating system (Linux or Solaris or other...)

The biggest advantage of a Beowulf cluster over massively parallel processors (MPPs) or supercomputers is the cost. Since inexpensive personal computers are used as nodes, a powerful Beowulf system can be built without spending a fortune. This cost advantage of ten can be as much as an order of magnitude over commercial systems of comparable capabilities. Another advantage of a Beowulf cluster is scalability. A wide range of system sizes is possible from a small number of nodes connected by a single low cost hub to system incorporating topologies of many hundreds of processors. These systems can be easily expanded over time as additional resources become available or extended requirements drive system size upward. The Beowulf is affordable, powerful, scalable, and easily expandable (Hawick, Grove, & Vaughan, 1999).

### *2.1.3 A General Cluster Architecture*

A cluster is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource. A computer node can be a single or multiprocessor system (PCs, workstations, or SMPs) with memory, I/O facilities, and an operating system. A cluster generally refers to two or more computers (nodes) connected together. The nodes can exist in a single cabinet or be physically separated and connected via a LAN. An interconnected (LAN-based) cluster of computers can appear as a single system to users and applications. Such a system can provide a cost-effective way to gain features and benefits (fast and reliable services) that have historically been found only on more expensive proprietary shared memory systems. The typical architecture of a cluster is shown in Figure 2.1 (Baker, & Buyya, 1999).

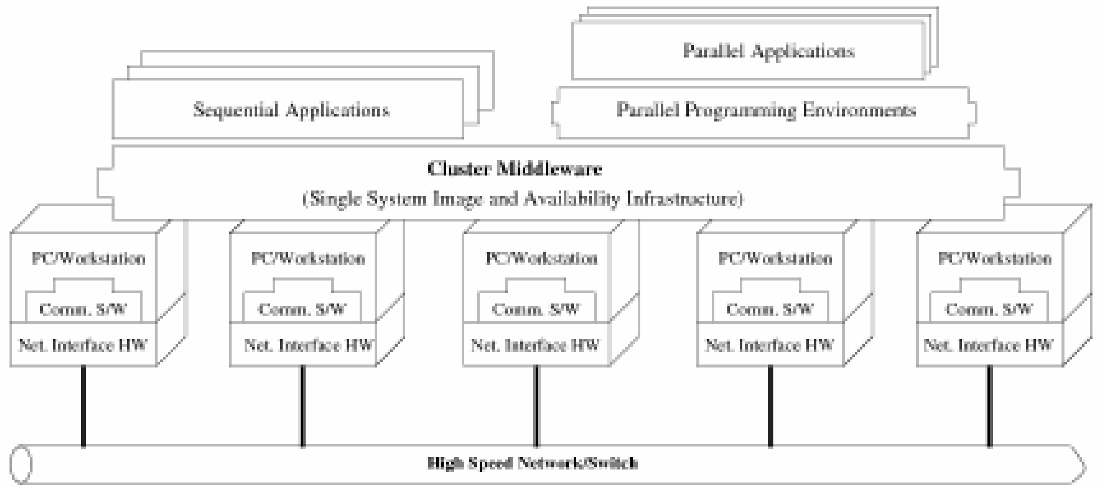

Parallel Applications
Sequential Applications
Parallel Programming Environments
Cluster Middleware
(Single System Image and Availability Infrastructure)
PC/Workstation
Comm. S/W
Net. Interface HW
High Speed Network/Switch

Figure 2.1 A general cluster architecture.

The following are some prominent components of cluster computers:

- Multiple High Performance Computers (PCs, Workstations, or SMPs)
- State-of-the-art Operating Systems (Layered or Micro-kernel based)
- High Performance Networks/Switches (such as Gigabit Ethernet and Myrinet)
- Network Interface Cards (NICs)
- Fast Communication Protocols and Services (such as Active and Fast Messages)
- Cluster Middleware (Single System Image (SSI) and System Availability Infrastructure)
    - Hardware (such as Digital (DEC) Memory Channel, hardware DSM, and SMP techniques)
    - Operating System Kernel or Gluing Layer (such as Solaris MC and GLUnix)
    - Applications and Subsystems
        - Applications (such as system management tools and electronic forms)
        - Runtime Systems (such as software DSM and parallel file system)

        - Resource Management and Scheduling software (such as LSF (Load Sharing Facility) and CODINE (COmputing in DIstributed Networked Environments))
- Parallel Programming Environments and Tools (such as compilers, PVM (Parallel Virtual Machine), and MPI (Message Passing Interface))
- Applications
    - Sequential
    - Parallel or Distributed

The network interface hardware acts as a communication processor and is responsible for transmitting and receiving packets of data between cluster nodes via a network/switch. Communication software offers a means of fast and reliable data communication among cluster nodes and to the outside world. Often, clusters with a special network/switch like Myrinet use communication protocols such as active messages for fast communication among its nodes. They potentially bypass the operating system and thus remove the critical communication overheads providing direct user-level access to the network interface.

The cluster nodes can work collectively, as an integrated computing resource, or they can operate as individual computers. The cluster middleware is responsible for offering an illusion of a unified system image (single system image) and availability out of a collection on independent but interconnected computers. Programming environments can offer portable, efficient, and easy-to-use tools for development of applications. They include message passing libraries, debuggers, and profilers. It should not be forgotten that clusters could be used for the execution of sequential or parallel applications.

### *2.1.4 Types of Clusters*

Clusters offer the following features at a relatively low cost (Baker, & Buyya, 1999) :

- High Performance
- Expandability and Scalability
- High Throughput
- High Availability

Cluster technology permits organizations to boost their processing power using standard technology (commodity hardware and software components) that can be acquired/purchased at a relatively low cost. This provides expandability--an affordable upgrade path that lets organizations increase their computing power--while preserving their existing investment and without incurring a lot of extra expenses. The performance of applications also improves with the support of scalable software environment. Another benefit of clustering is a failover capability that allows a backup computer to take over the tasks of a failed computer located in its cluster. Clusters are classified into many categories based on various factors as indicated below (Baker, & Buyya, 1999).

**1. Application Target** - Computational science or mission-critical applications.

- High Performance Clusters (HPCs)
- High Availability (HA) Clusters
- Load Balancing (LB) Clusters

**2. Node Ownership** - Owned by an individual or dedicated as a cluster node.

- Dedicated Clusters
- Nondedicated Clusters

The distinction between these two cases is based on the ownership of the nodes in a cluster. In the case of dedicated clusters, a particular individual does not own a workstation; the resources are shared so that parallel computing can be performed across the entire cluster. The alternative nondedicated case is where individuals own workstations and applications are executed by stealing idle CPU cycles. The motivation for this scenario is based on the fact that most workstation CPU cycles are unused, even during peak hours. Parallel computing on a dynamically changing set of nondedicated workstations is called adaptive parallel computing. In nondedicated clusters, a tension exists between the workstation owners and remote users who need the workstations to run their application. The former expects fast interactive response from their workstation, while the latter is only concerned with fast application turnaround by utilizing any spare CPU cycles. This emphasis on sharing the processing resources erodes the concept of node ownership and introduces the need for complexities such as process migration and load balancing strategies. Such strategies allow clusters to deliver adequate interactive performance as well as to provide shared resources to demanding sequential and parallel applications.

**3. Node Hardware** - PC, Workstation, or SMP.

- Clusters of PCs (CoPs) or Piles of PCs (PoPs)
- Clusters of Workstations (COWs)
- Clusters of SMPs (CLUMPs)

**4. Node Operating System** - Linux, Windows, Solaris, AIX, etc.

- Linux Clusters (e.g., Beowulf)
- Solaris Clusters (e.g., Berkeley NOW)
- Microsoft Clusters (e.g., HPVM)
- AIX Clusters (e.g., IBM SP2)
- HP-UX clusters

**5. Node Configuration** - Node architecture and type of OS it is loaded with.

- Homogeneous Clusters: All nodes will have similar architectures and run the same OSs.
- Heterogeneous Clusters: All nodes will have different architectures and run different OSs.

**6. Levels of Clustering** - Based on location of nodes and their count.

- Group Clusters (#nodes: 2-99): Nodes are connected by SANs (System Area Networks) like Myrinet and they are either stacked into a frame or exist within a center.
- Departmental Clusters (#nodes: 10s to 100s)
- Organizational Clusters (#nodes: many 100s)
- National Metacomputers (WAN/Internet-based): (#nodes: many departmental / organizational systems or clusters)
- International Metacomputers (Internet-based): (#nodes: 1000s to many millions)

Individual clusters may be interconnected to form a larger system (clusters of clusters) and, in fact, the Internet itself can be used as a computing cluster. The use of wide-area networks of computer resources for high performance computing has led to the emergence of a new field called Metacomputing.

## 2.2 Load Balancing In Distributed Systems

### *2.2.1 The Concept of Load Distribution*

A distributed system consists of a collection of autonomous computers connected by a local area communication network. Users submit tasks at their host computers for processing. As Figure 2.2 shows, the random arrival of tasks in such an environment can cause some computers to be heavily loaded while other computers are idle or only lightly loaded. Load distributing improves performance by transferring tasks from heavily loaded computers, where service is poor, to lightly loaded computers, where the tasks can take advantage of computing capacity that would otherwise go unused (Shivaratri, Krueger, & Singhal, 1992).

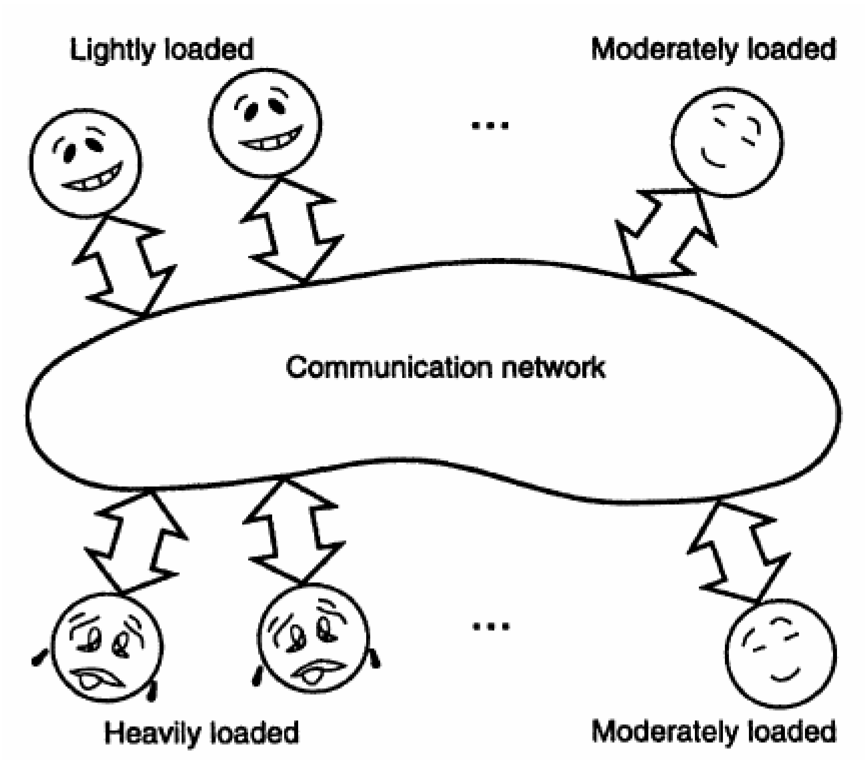


Figure 2.2 A system without load distribution (Shivaratri, Krueger, & Singhal, 1992).

If workloads at some computers are typically heavier than at others, or if some processors execute tasks more slowly than others, the situation shown in Figure 1 is likely to occur often. The usefulness of load distributing is not so obvious in systems in which all processors are equally powerful and have equally heavy workloads over the long term. However, Livny and Melman (1982) have shown that even in such a homogeneous distributed system, at least one computer is likely to be idle while other computers are heavily loaded because of statistical fluctuations in the arrival of tasks to computers and task-service-time requirements. Therefore, even in a homogeneous distributed system, system performance can potentially be improved by appropriate transfers of workload from heavily loaded computers (senders) to idle or lightly loaded computers (receivers).

A widely used performance metric is the average response time of tasks. The response time of a task is the time elapsed between its initiation and its completion. Minimizing the average response time is often the goal of load distribution.

A key issue in the design of dynamic load-distributing algorithms is identifying a suitable load index. A load index predicts the performance of a task if it is executed at some particular node. To be effective, load index readings taken when tasks initiate should correlate well with task-response times. Load indexes that have been studied and used include the length of the CPU queue, the average CPU queue length over some period, the amount of available memory, the context-switch rate, the system call rate, and CPU utilization. Researchers have consistently found significant differences in the effectiveness of such load indexes — and that simple load indexes are particularly effective. For example, Kunz (1991) found that the choice of a load index has considerable effect on performance, and that the most effective of the indexes we have mentioned is the CPU queue length. Furthermore, Kunz found no performance improvement over this simple measure when combinations of these load indexes were used. It is crucial that the mechanism used to measure load be efficient and impose minimal overhead.

### *2.2.2 The Classification of Distributed Scheduling*

The operating system and management of the concurrent processes constitute integral parts of the parallel and distributed environments. One of the biggest issues in such systems is the development of effective techniques for the distribution of the processes of a parallel program on multiple processors. The problem is how to distribute (or schedule) the processes among processing elements to achieve some performance goal(s), such as minimizing execution time, minimizing communication delays, and/or maximizing resource utilization (Shirazi, Husson, & Kavi, 1995). Process scheduling methods are typically classified into several subcategories (Casavant, Kuhl, 1988) as depicted in Figure 2.3.

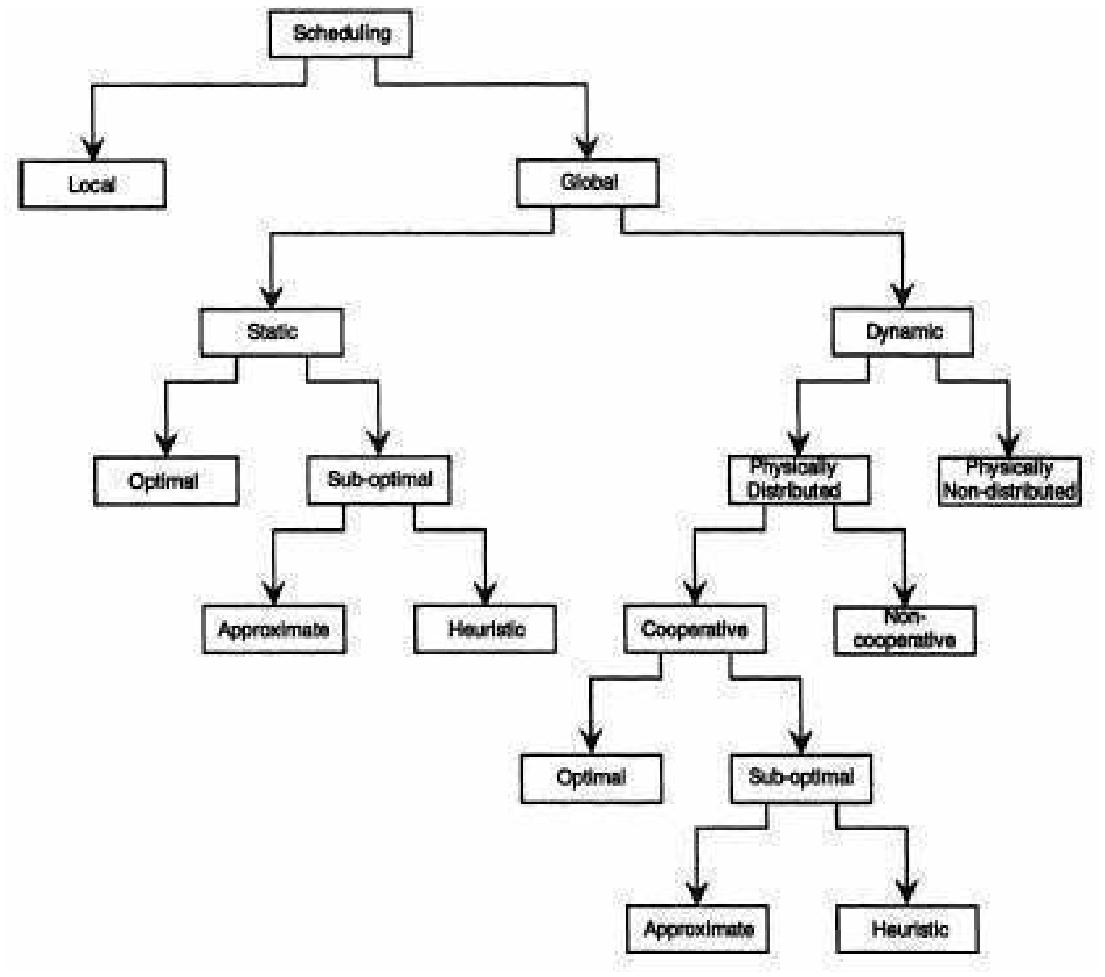


Figure 2.3 Classification of scheduling methods (Casavant, Kuhl, 1988).

**a) Local Versus Global:** At the highest level, we may distinguish between local and global scheduling. Local scheduling is involved with the assignment of processes to the time-slices of a single processor. Since the area of scheduling on single-processor systems, as well as the area of sequencing or job-shop scheduling, has been actively studied for a number of years, this taxonomy will focus on global scheduling. Global scheduling is the problem of deciding where to execute a process, and the job of local scheduling is left to the operating system of the processor to which the process is ultimately allocated. This allows the processors in a multiprocessor increased autonomy while reducing the responsibility (and consequently overhead) of the global scheduling mechanism. Note that this does not imply that global scheduling must be done by a single central authority, but rather, we view the problems of local and global scheduling as separate issues, and (at least logically) separate mechanisms are at work solving each.

**b) Static Versus Dynamic:** The next level in the hierarchy (beneath global scheduling) is a choice between static and dynamic scheduling. This choice indicates the time at which the scheduling or assignment decisions are made.

In the case of static scheduling, information regarding the total mix of processes in the system as well as all the independent subtasks involved in a job or task force, is assumed to be available by the time the program object modules are linked into load modules. Hence, each executable image in a system has a static assignment to a particular processor, and each time that process image is submitted for execution, it is assigned to that processor. A more relaxed definition of static scheduling may include algorithms that schedule task forces for a particular hardware configuration. Over a period of time, the topology of the system may change, but characteristics describing the task force remain the same. Hence, the scheduler may generate a new assignment of processes to processors to serve as the schedule until the topology changes again.

**c) Optimal Versus Sub optimal:** In the case that all information regarding the state of the system as well as the resource needs of a process are known, an optimal assignment can be made based on some criterion function. Examples of optimization measures are minimizing total process completion time, maximizing utilization of resources in the system, or maximizing system throughput. In the event that these problems are computationally infeasible, suboptimal solutions may be tried. Within the realm of suboptimal solutions to the scheduling problem, we may think of two general categories.

**d) Approximate Versus Heuristic:** The first is to use the same formal computational model for the algorithm, but instead of searching the entire solution space for an optimal solution, we are satisfied when we find a "good" one. We will categorize these solutions as suboptimal-approximate. The assumption that a good solution can be recognized may not be so insignificant, but in the cases where a metric is available for evaluating a solution, this technique can be used to decrease the time taken to find an acceptable solution (schedule). The factors which determine whether this approach is worthy of pursuit include:

- Availability of a function to evaluate a solution.
- The time required to evaluate a solution.
- The ability to judge according to some metric the value of an optimal solution.
- Availability of a mechanism for intelligently pruning the solution space.

The second branch beneath the suboptimal category is labeled heuristic. This branch represents the category of static algorithms which make the most realistic assumptions about a priori knowledge concerning process and system loading characteristics. It also represents the solutions to the static scheduling problem which require the most reasonable amount of time and other system resources to perform their function. The most distinguishing feature of heuristic schedulers is that they make use of special parameters which affect the system in indirect ways. Often, the parameter being monitored is correlated to system performance in an indirect instead of a direct way, and this alternate parameter is much simpler to monitor or calculate. For example, clustering groups of processes which communicate heavily on the same processor and physically separating processes which would benefit from parallelism directly decreases the overhead involved in passing information between processors, while reducing the interference among processes which may run without synchronization with one another. This result has an impact on the overall service that users receive, but cannot be directly related (in a quantitative way) to system performance as the user sees it. Hence, our intuition, if nothing else, leads us to believe that taking the aforementioned actions when possible will improve system performance. However, we may not be able to prove that a first-order relationship between the mechanism employed and the desired result exists (Casavant, Kuhl, 1988).

**e) Optimal and Suboptimal Approximate Techniques:** Regardless of whether a static solution is optimal or suboptimal-approximate, there are four basic categories of task allocation algorithms which can be used to arrive at an assignment of processes to processors.

- Solution space enumeration and search.
- Graph theoretic.
- Mathematical programming.
- Queueing theoretic.

**f) Dynamic Solutions:** In the dynamic scheduling problem, the more realistic assumption is made that very little a priori knowledge is available about the resource needs of a process. It is also unknown in what environment the process will execute during its lifetime. In the static case, a decision is made for a process image before it is ever executed, while in the dynamic case no decision is made until a process begins its life in the dynamic environment of the system. Since it is the responsibility of the running system to decide where a process is to execute, it is only natural to next ask where the decision itself is to be made.

**g) Distributed Versus Nondistributed:** The next issue (beneath dynamic solutions) involves whether the responsibility for the task of global dynamic scheduling should physically reside in a single processor (physically nondistributed) or whether the work involved in making decisions should be physically distributed among the processors. Here the concern is with the logical authority of the decision-making process.

**h) Cooperative Versus Noncooperative:** Within the realm of distributed dynamic global scheduling, we may also distinguish between those mechanisms which involve cooperation between the distributed components (cooperative) and those in which the individual processors make decisions independent of the actions of the other processors (noncooperative). The question here is one of the degree of autonomy which each processor has in determining how its own resources should be used. In the noncooperative case individual processors act alone as autonomous entities and arrive at decisions regarding the use of their resources independent of the effect of their decision on the rest of the system. In the cooperative case each processor has the responsibility to carry out its own portion of the scheduling task, but all processors are working toward a common system wide goal. In other words, each processor's local operating system is concerned with making decisions in concert with the other processors in the system in order to achieve some global goal, instead of making decisions based on the way in which the decision will affect local performance only. As in the static case, the taxonomy tree has reached a point where we may consider optimal, suboptimal-approximate, and suboptimal-heuristic solutions. The same discussion as was presented for the static case applies here as well (Casavant, Kuhl, 1988).

In addition to the hierarchical portion of the taxonomy already discussed, there are a number of other distinguishing characteristics which scheduling systems may have. The following sections will deal with characteristics which do not fit uniquely under any particular branch of the tree-structured taxonomy given thus far, but are still important in the way that they describe the behavior of a scheduler. In other words, the following could be branches beneath several of the leaves shown in Fig. 2 and in the interest of clarity are not repeated under each leaf, but are presented here as a flat extension to the scheme presented thus far. It should be noted that these attributes represent a set of characteristics, and any particular scheduling subsystem may possess some subset of this set. Finally, the placement of these characteristics near the bottom of the tree is not intended to be an indication of their relative importance or any other relation to other categories of the hierarchical portion. Their position was determined primarily to reduce the size of the description of the taxonomy.

### *2.2.3 Components of A Load Distribution Algorithm*

Typically, a dynamic load distributing algorithm has four components: a transfer policy, a selection policy, a location policy, and an information policy (Shivaratri, Krueger, & Singhal, 1992).

**a) Transfer policy:** A transfer policy determines whether a node is in a suitable state to participate in a task transfer, either as a sender or a receiver. Many proposed transfer policies are threshold policies. Thresholds are expressed in units of load. When a new task originates at a node, the transfer policy decides that the node is a sender if the load at that node exceeds a threshold T1. On the other hand, if the load at a node falls below T2, the transfer policy decides that the node can be a receiver for a remote task. Depending on the algorithm, T, and T2 may or may not have the same value.

Alternatives to threshold transfer policies include relative transfer policies. Relative policies consider the load of a node in relation to loads at other system nodes. For example, a relative policy might consider a node to be a suitable receiver if its load is lower than that of some other node by at least some fixed value. Alternatively, a node might be considered a receiver if its load is among the lowest in the system.

**b) Selection policy:** Once the transfer policy decides that a node is a sender, a selection policy selects a task for transfer. Should the selection policy fail to find a suitable task to transfer, the node is no longer considered a sender. The simplest approach is to select one of the newly originated tasks that caused the node to become a sender. Such a task is relatively cheap to transfer, since the transfer is nonpreemptive. A selection policy considers several factors in selecting a task:

1) The overhead incurred by the transfer should be minimal. For example, a small task carries less overhead.

2) The selected task should be long lived so that it is worthwhile to incur the transfer overhead.

3) The number of location-dependent system calls made by the selected task should be minimal. Location-dependent calls are system calls that must be executed

on the node where the task originated, because they use resources such as windows, the clock, or the mouse that are only at that node.

**c) Location policy:** The location policy's responsibility is to find a suitable "transfer partner" (sender or receiver) for a node, once the transfer policy has decided that the node is a sender or receiver. A widely used decentralized policy finds a suitable node through polling: A node polls another node to find out whether it is suitable for load sharing. Nodes can be polled either serially or in parallel (for example, multicast). A node can be selected for polling on a random basis, on the basis of the information collected during the previous polls, or on a nearest neighbor basis. An alternative to polling is to broadcast a query seeking any node available for load sharing. In a centralized policy, a node contacts one specified node called a coordinator to locate a suitable node for load sharing. The coordinator collects information about the system (which is the responsibility of the information policy), and the transfer policy uses this information at the coordinator to select receivers.

**d) Information policy:** The information policy decides when information about the states of other nodes in the system is to be collected, from where it is to be collected, and what information is collected. There are three types of information policies:

*1) Demand-driven policies:* Under these decentralized policies, a node collects the state of other nodes only when it becomes either a sender or a receiver, making it a suitable candidate to initiate load sharing. A demand-driven information policy is inherently a dynamic policy, as its actions depend on the system state. Demand-driven policies may be sender, receiver, or symmetrically initiated. In sender-initiated policies, senders look for receivers to which they can transfer their load. In receiver-initiated policies, receivers solicit loads from senders. A symmetrically initiated policy is a combination of both: Load-sharing actions are triggered by the demand for extra processing power or extra work.

*2) Periodic policies:* These policies, which may be either centralized or decentralized, collect information periodically. Depending on the information collected, the transfer policy may decide to transfer tasks. Periodic information policies Generally do not adapt their rate of activity to the system state. For example, the benefits resulting from load distributing are minimal at high system loads because most nodes in the system are busy. Nevertheless, overheads due to periodic information collection continue to increase the system load and thus worsen the situation.

*3) State-change-driven policies:* Under state-change-driven policies, nodes disseminate information about their states whenever their states change by a certain degree. A state-change-driven policy differs from a demand-driven policy in that it disseminates information about the state of a node, rather than collecting information about other nodes. Under centralized state-change driven policies, nodes send state information to a centralized collection point. Under decentralized state-change driven policies, nodes send information to peers.

### ***2.2.4 Load Distribution Algorithms***

#### *2.2.4.1 Sender-initiated algorithms.*

Under sender-initiated algorithms, load-distributing activity is initiated by an overloaded node (sender) trying to send a task to an underloaded node (receiver) (Shivaratri, Krueger, & Singhal, 1992).

**Transfer policy:** Each of the algorithms uses the same transfer policy, a threshold policy based on the CPU queue length. A node is identified as a sender if a new task originating at the node makes the queue length exceed a threshold T. A node identifies itself as a suitable receiver for a task transfer if accepting the task will not cause the node's queue length to exceed T. Selection policy. All three algorithms have the same selection policy, considering only newly arrived tasks for transfer.

**Location policy:** The algorithms differ only in their location policies, which we review in the following subsections.

*a) Random:* One algorithm has a simple dynamic location policy called random, which uses no remote state information. A task is simply transferred to a node selected at random, with no information exchange between the nodes to aid in making the decision. Useless task transfers can occur when a task is transferred to a node that is already heavily loaded (its queue length exceeds). An issue is how a node should treat a transferred task. If a transferred task is treated as a new arrival, then it can again be transferred to another node, providing the local queue length exceeds T. If such is the case, then irrespective of the average load of the system, the system will eventually enter a state in which the nodes are spending all their time transferring tasks, with no time spent executing them. A simple solution is to limit the number of times a task can be transferred. Despite its simplicity, this random location policy provides substantial performance improvements over systems not using load distributing.

*b) Threshold:* A location policy can avoid useless task transfers by polling a node (selected at random) to determine whether transferring a task would make its queue length exceed T. If not, the task is transferred to the selected node, which must execute the task regardless of its state when the task actually arrives. Otherwise, another node is selected at random and is polled. To keep the overhead low, the number of polls is limited by a parameter called the poll limit. If no suitable receiver node is found within the poll limit polls, then the node at which the task originated must execute the task. By avoiding useless task transfers, the threshold policy provides a substantial performance improvement over the random location policy.

*c) Shortest:* The two previous approaches make no effort to choose the best destination node for a task. Under the shortest location policy, a number of nodes (poll limit) are selected at random and polled to determine their queue length. The node with the shortest queue is selected as the destination for task transfer, unless its queue length is greater than or equal to T. The destination node will execute the task regardless of its queue length when the transferred task arrives. The performance improvement obtained by using the shortest location policy over the threshold policy was found to be marginal, indicating that using more detailed state information does not necessarily improve system performance significantly.

**Information policy:** When either the shortest or the threshold location policy is used, polling starts when the transfer policy identifies a node as the sender of a task. Hence, the information policy is demand driven.

Sender-initiated algorithms using any of the three location policies cause system instability at high system loads. At such loads, no node is likely to be lightly loaded, so a sender is unlikely to find a suitable destination node. However, the polling activity in sender-initiated algorithms increases as the task arrival rate increases, eventually reaching a point where the cost of load sharing is greater than its benefit. At a more extreme point, the workload that cannot be offloaded from a node, together with the overhead incurred by polling, exceeds the node's CPU capacity and instability results. Thus, he actions of sender-initiated algorithms are not effective at high system loads and cause system instability, because the algorithms fail to adapt to the system state.

### *2.2.4.2 Receiver-initiated algorithms*

In receiver-initiated algorithms, load distributing activity is initiated from an underloaded node (receiver), which tries to get a task from an overloaded node (sender) (Shivaratri, Krueger, & Singhal, 1992).

**Transfer policy:** The algorithm's threshold transfer policy bases its decision on the CPU queue length. The policy is triggered when a task departs. If the local queue length falls below the threshold T then the node is identified as a receiver for obtaining a task from a node (sender) to be determined by the location policy. A node is identified to be a sender if its queue length exceeds the threshold T.

**Selection policy:** The algorithm considers all tasks for load distributing, and can use any of the approaches discussed before.

**Location policy:** The location policy selects a node at random and polls it to determine whether transferring a task would place its queue length below the threshold level. If not, then the polled node transfers a task. Otherwise, another node is selected at random, and the procedure is repeated until either a node that can transfer a task (a sender) is found or a static poll limit number of tries has failed to find a sender. A problem with the location policy is that if all polls fail to find a sender, then the processing power available at a receiver is completely lost by the system until another task originates locally at the receiver (which may not happen for a long time). The problem severely affects performance in systems where only a few nodes generate most of the system workload and random polling by receivers can easily miss them. The remedy is simple: If all the polls fail to find a sender, then the node waits until another task departs or for a predetermined period before reinitiating the load distributing activity, provided the node is still a receiver.

**Information policy:** The information policy is demand driven, since polling starts only after a node becomes a receiver.

Receiver-initiated algorithms do not cause system instability because, at high system loads, a receiver is likely to find a suitable sender within a few polls. Consequently, polls are increasingly effective with increasing system load, and little waste of CPU capacity results.

Under the most widely used CPU scheduling disciplines (such as round-robin and its variants), a newly arrived task is quickly provided a quantum of service. In receiver-initiated algorithms, the polling starts when a node becomes a receiver. However, these polls seldom arrive at senders just after new tasks have arrived at the senders but before these tasks have begun executing. Consequently, most transfers are preemptive and therefore expensive. Sender-initiated algorithms, on the other hand, make greater use of nonpreemptive transfers, since they can initiate load-distributing activity as soon as a new task arrives. An alternative to this receiver-initiated algorithm is the reservation algorithm. Rather than negotiate an immediate transfer, a receiver requests that the next task to arrive be nonpreemptively transferred. Upon arrival, the "reserved" task is transferred to the receiver if the receiver is still a receiver at that time. While this algorithm does not require preemptive task transfers, it was found to perform significantly worse than the sender initiated algorithms.

#### *2.2.4.3 Symmetrically initiated algorithms*

Under symmetrically initiated algorithms, 10 both senders and receivers initiate load-distributing activities for task transfers. These algorithms have the advantages of both sender and receiver initiated algorithms. At low system loads, the sender-initiated component is more successful at finding underloaded nodes. At high system loads, the receiver-initiated component is more successful at finding overloaded nodes. However, these algorithms may also have the disadvantages of both sender and receiver-initiated algorithms. As with sender-initiated algorithms, polling at high system loads may result in system instability. As with receiver initiated algorithms, a preemptive task transfer facility is necessary. A simple symmetrically initiated algorithm can be constructed by combining the transfer and location policies described for sender-initiated and receiver-initiated algorithms (Shivaratri, Krueger, & Singhal, 1992).

#### *2.2.4.4 Adaptive algorithms*

A stable symmetrically initiated adaptive algorithm. The main cause of system instability due to load sharing in the previously reviewed algorithms is indiscriminate polling by the sender's negotiation component. The stable symmetrically initiated algorithm uses the information gathered during polling (instead of discarding it, as the previous algorithms do) to classify the nodes in the system as sender/overloaded, receiver/underloaded, or OK (nodes having manageable load). The knowledge about the state of nodes is maintained at each node by a data structure composed of a senders list, a receivers list, and an OK list. These lists are maintained using an efficient scheme: List-manipulative actions, such as moving a node from one list to another or determining to which list a node belongs, impose a small and constant overhead, irrespective of the number of nodes in the system. Consequently, this algorithm scales well to large distributed systems (Shivaratri, Krueger, & Singhal, 1992).

Initially, each node assumes that every other node is a receiver. This state is represented at each node by a receivers list containing all nodes (except the node itself), and an empty senders list and OK list.

**Transfer policy:** The threshold transfer policy makes decisions based on the CPU queue length. The transfer policy is triggered when a new task originates or when a task departs. The policy uses two threshold values — a lower threshold and an upper threshold—to classify the nodes. A node is a sender if its queue length is greater than its upper threshold, a receiver if its queue length is less than its lower threshold, and OK otherwise.

**Location policy:** The location policy has two components: the sender-initiated component and the receiver-initiated component. The sender-initiated component is triggered at a node when it becomes a sender. The sender polls the node at the head of the receivers list to determine whether it is still a receiver. The polled node removes the sender node ID from the list it is presently in, puts it at the head of its senders list, and informs the sender whether it is currently a receiver, sender, or OK. On receipt of this reply, the sender transfers the new task if the polled node has indicated that it is a receiver. Otherwise, the polled node's ID is removed from the receivers list and is put at the head of the OK list or the senders list based on its reply.

Polling stops if a suitable receiver is found for the newly arrived task, if the number of polls reaches a poll limit (a parameter of the algorithm), or if the receivers list at the sender node becomes empty. If polling fails to find a receiver, the task is processed locally, though it may later be preemptively transferred as a result of receiver-initiated load sharing. The goal of the receiver-initiated component is to obtain tasks from a sender node. The nodes polled are selected in the following order:

1) Head to tail in the senders list. The most up-to-date information is used first.

2) Tail to head in the OK list. The most out-of-date information is used first in the hope that the node has become a sender.

3) Tail to head in the receivers list. Again, the most out-of-date information is used first.

The receiver-initiated component is triggered at a node when the node becomes a receiver. The receiver polls the selected node to determine whether it is a sender. On receipt of the message, the polled node, if it is a sender, transfers a task to the polling node and informs it of its state after the task transfer. If the polled node is not a sender, it removes the receiver node ID from the list it is presently in, puts it at the head of the receivers list, and informs the receiver whether the polled node is a receiver or OK. On receipt of this reply, the receiver node removes the polled node ID from whatever list it is presently in and puts it at the head of its receivers list or OK list, based on its reply. Polling stops if a sender is found, if the receiver is no longer a receiver, or if the number of polls reaches a static poll limit.

**Selection policy:** The sender-initiated component considers only newly arrived tasks for transfer. The receiver-initiated component can use any of the approaches discussed before.

**Information policy:** The information policy is demand driven, as polling starts when a node becomes either a sender or a receiver.

At high system loads, the probability of a node's being underloaded is negligible, resulting in unsuccessful polls by the sender-initiated component. Unsuccessful polls result in the removal of polled node IDs from receivers lists. Unless receiver-initiated polls to these nodes fail to find senders, which is unlikely at high system loads, the receivers lists remain empty. This scheme prevents future sender-initiated polls at high system loads (which are most likely to fail). Hence, the sender-initiated component is deactivated at high system loads, leaving only receiver-initiated load sharing (which is effective at such loads). At low system loads, receiver-initiated polls are frequent and generally fail. These failures do not adversely affect performance, since extra processing capacity is available at low system loads.

In addition, these polls have the positive effect of updating the receivers lists. With the receivers lists accurately reflecting the system's state, future sender-initiated load sharing will generally succeed within a few polls. Thus, by using sender-initiated load sharing at low system loads, receiver-initiated load sharing at high loads, and symmetrically initiated load sharing at moderate loads, the stable symmetrically initiated algorithm achieves improved performance over a wide range of system loads and preserves system stability.

#### *2.2.4.5 A stable sender-initiated adaptive algorithm*

This algorithm uses the sender-initiated load-sharing component of the previous approach but has a modified receiver-initiated component to attract future nonpreemptive task transfers from sender nodes. An important feature is that the algorithm performs load sharing only with nonpreemptive transfers, which are cheaper than preemptive transfers. The stable sender initiated algorithm is very similar to the stable symmetrically initiated algorithm. In the following, we point out only the differences (Shivaratri, Krueger, & Singhal, 1992).

In the stable sender-initiated algorithm, the data structure (at each node) of the stable symmetrically initiated algorithm is augmented by an array called the state vector. Each node uses the state vector to keep track of which list (senders, receivers, or OK) it belongs to at all the other nodes in the system. For example, state vector[nodeid] says to which list node i belongs at the node indicated by nodeid. As in the stable symmetrically initiated algorithm, the overhead for maintaining this data structure is small and constant, irrespective of the number of nodes in the system. The sender-initiated load sharing is augmented with the following step:

When a sender polls a selected node, the sender's state vector is updated to show that the sender now belongs to the senders list at the selected node. Likewise, the polled node updates its state vector based on the reply it sent to the sender node to reflect which list it will belong to at the sender.

The receiver-initiated component is replaced by the following protocol:

When a node becomes a receiver, it informs only those nodes that are misinformed about its current state. The misinformed nodes are those nodes whose receivers lists do not contain the receiver's ID. This information is available in the state vector at the receiver. The state vector at the receiver is then updated to reflect that it now belongs to the receivers list at all those nodes that were misinformed about its current state. There are no preemptive transfers of partly executed tasks here. The sender initiated load-sharing component will do any task transfers, if possible, on the arrival of a new task. The reasons for this algorithm's stability are the same as for the stable symmetrically initiated algorithm (Shivaratri, Krueger, & Singhal, 1992).

# CHAPTER THREE
# THE CLUSTER INFRASTRUCTURE MODEL

There can be tens to hundreds of computers in a Beowulf cluster. A subsystem is needed to manage the nodes in a scalable manner to provide a stable and reliable distributed computing system. The cluster infrastructure model (CIM) was designed with this objective. CIM is responsible for maintaining the components of the cluster by keeping the records of active nodes, checking their health and isolating failed nodes. CIM also serves as an information service for the distributed load balancing model by collecting the state information from the nodes.

## 3.1 CIM Architecture

CIM is designed as a hierarchically centralized model. There are three components in CIM. On the top there is a Cluster Manager (CM) which constructs the cluster and manages the resources and components of the whole cluster. It does its job via a number of  low level managers called Node Managers. A Node Manager (NM) is responsible for managing a discrete subset of the nodes of the cluster. A Node (N) represents a single processing unit, a worker of the cluster, such as an independent workstation. Each N has its own resources like memory, CPU, disk, etc. to be managed. CM assigns each joined N to a an NM. To manage Ns in a scalable manner, CM determines the number of NMs according to the number of Ns. As number of Ns increases CM employs new NMs. The hierarchically centralized architecture of CIM is shown in Figure 3.1.

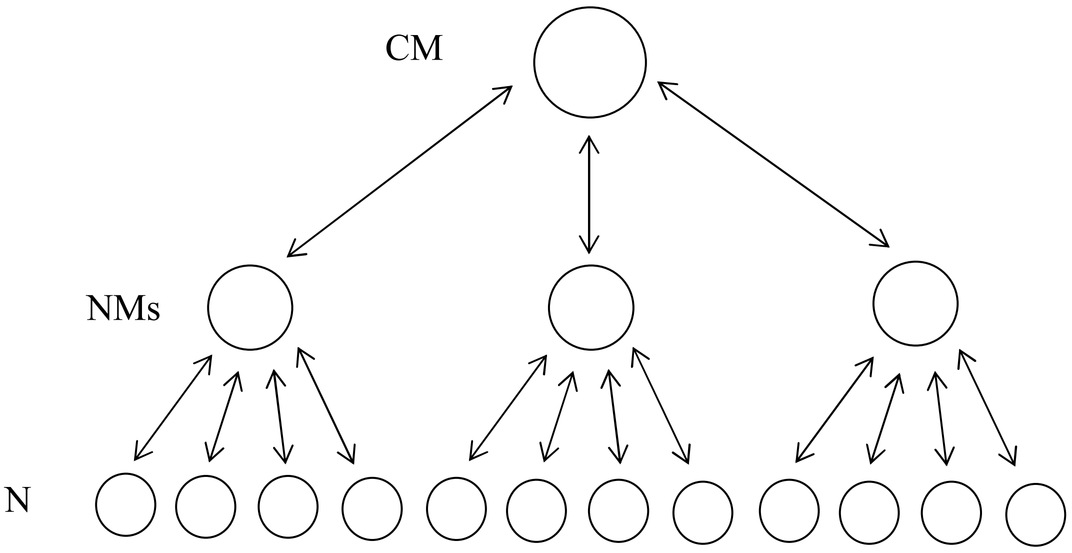


Figure 3.1 CIM architecture.

## 3.2 Communication in CIM

CM, NMs and Ns are communicated with point-to-point and also multicast messages. Each NM has a multicast group consisting of its Ns. Also CM has a multicast group that has members of NMs. Group communications are performed by these multicast groups. Multicast group structure is shown in Figure 3.2.

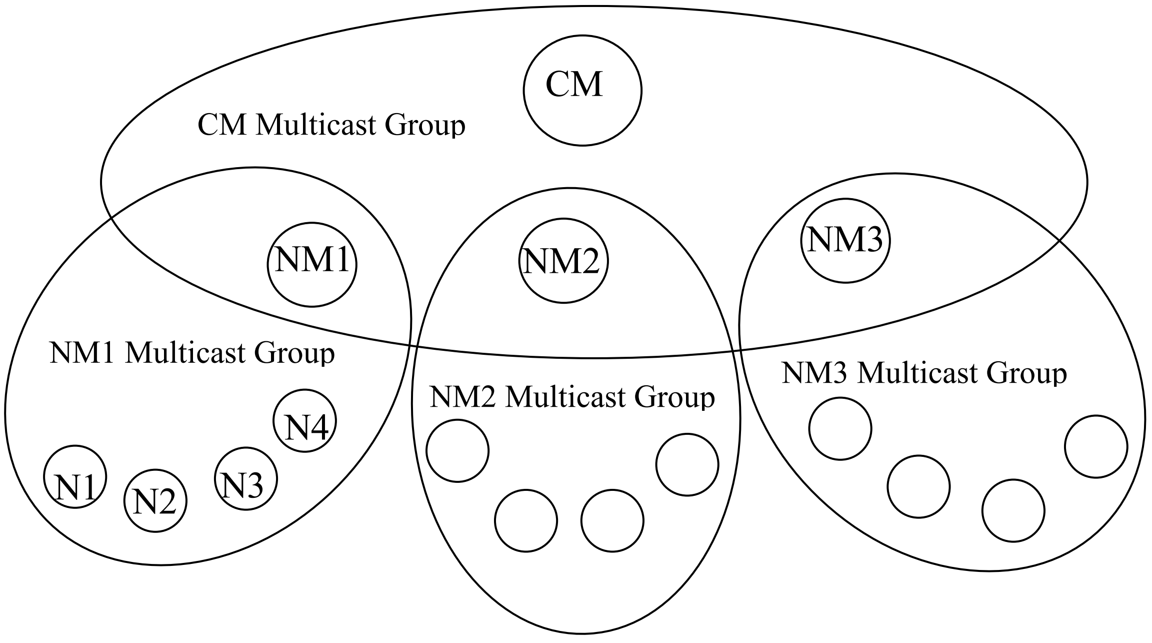


Figure 3.2 Multicast groups in CIM.

## 3.3 Fault Tolerance in CIM

The CM is responsible for the health of the cluster. NMs are responsible for the health of its group of Ns. If a N is communicated with its NM with some reason in a specific time interval called Heartbeat-time, the NM knows that its N is alive. For this reason, a N, that is not communicated with its NM in Heartbeat-time interval, sends an Heartbeat (HB) message to tell NM that it is alive. By this way NM checks the health of its Ns. NM sends a CheckAlive message to a suspected N, which is not communicated with it in Heartbeat-time period to reply immediately with a HB message. When NM doesn't get a reply, it understands that the N is dead, and immediately informs CM to remove it from the cluster.

In a similar way, CM checks aliveness of NMs. An NM sends HB message to CM if it is not communicated with it in HB-time period. CM waits a reply to its CheckAlive message sent to a suspected NM. CM promotes an N (the backup NM) as the new NM of the group of a dead NM. Each NM has a backup NM in its group. Also CM has a backup which checks the health of the CM. If the backup CM determines that the CM is dead, it promotes itself as the new CM.

Generally the backup NM is the second N in the group and the backup CM is the second NM in the cluster. As shown in figure 3, CM, NM, and N components are considered as processes or threads running on a workstation of the cluster. The first workstation runs the CM, an NM, and an N processes at the same time. If that workstation crashes, all running N, NM and CM processes are dead as well. Considering this situation, second NM which is running on a different workstation is chosen as backup of CM. Similarly the workstation running the first N of a node group also has the NM of that group, so the second N is chosen as the backup NM.

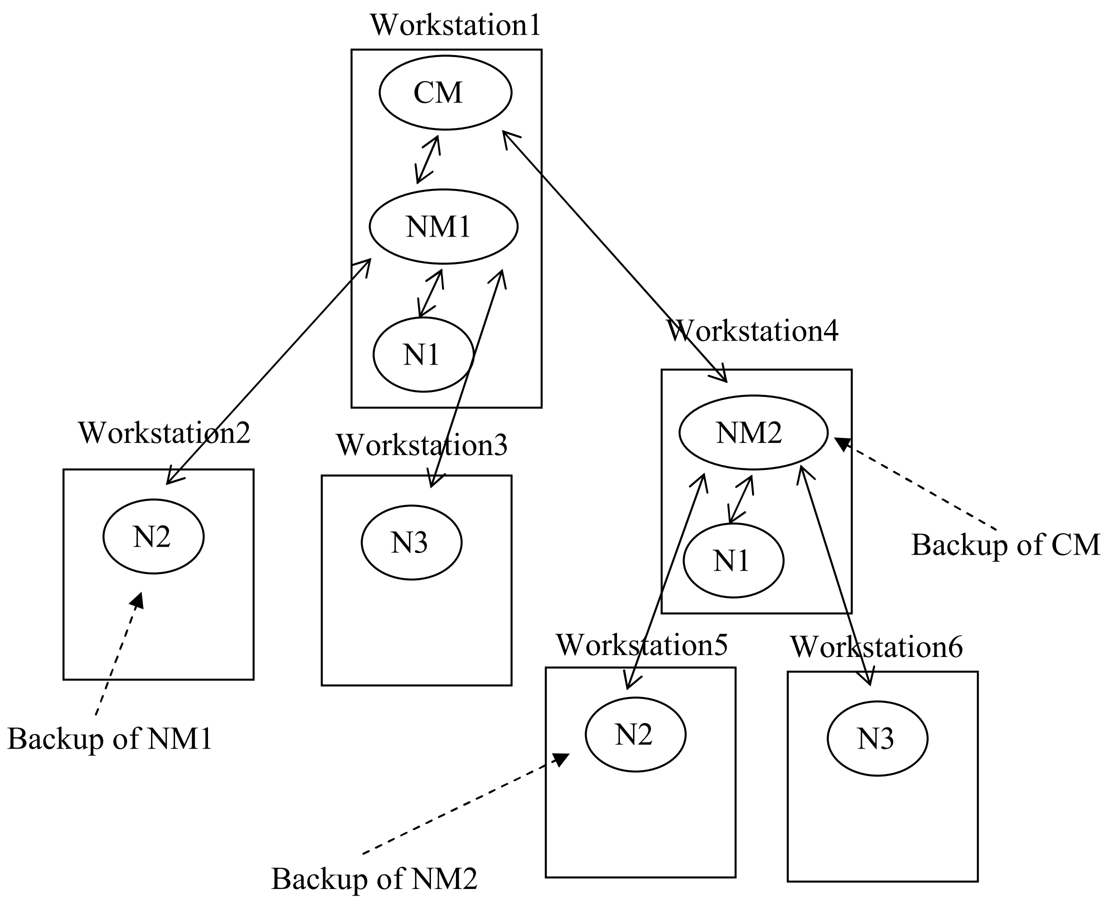


Figure 3.3 Backup structure of CIM.

## 3.4 Formal Protocol Design of CIM

The CIM protocol is designed using finite state machines (FSM). In a FSM; states, incoming events and outgoing events are defined. Then operations performed in occurance of incoming events at related states are described in an event-state table. After that the state transition diagram is generated which shows state changes according to occurance of incoming events. Finally pseudocodes are generated by using designed FSMs.

CM, NM and N components are designed as different modules, so their FSM are designed separately. In implementation these modules can be separate processes or threads.

### *3.4.1 Finite State Machines*

#### *3.4.1.1 Cluster Manager*

| **Incoming Events:** | **Outgoing Events:** |
|---|---|
| 1 StartCM_msg_received | HB_msg_sent |
| 2 Startup_timeout | CheckAlive_msg_sent |
| 3 NMOK_msg_received | StartNM_msg_sent |
| 4 HB_msg_received | AddN_msg_sent |
| 5 HB_timeout | SPLIT_msg_sent |
| 6 Ncrash_msg_received | NewCM_msg_sent |
| 7 NJoin_msg_received | |
| 8 Split_timeout | **States:** |
| 9 NAdded_msg_received | Startup |
| 10 CheckAlive_timeout | Ready |
| 11 Split_condition_occured | CrashDetect |
| 12 CheckAlive_msg_received | Split |
| 13 Activity_timeout | |

Table 3.1 Event-state table of CM module

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NULL | a | - | - | - | - | - | - | - | - | - | - | - | - |
| Startup | - | b | c | d | 0 | 0 | 0 | - | 0 | - | - | m | m |
| Ready | - | - | - | d | e | f | g | - | i | - | j | m | m |
| CrashDetect | - | - | - | k | 0 | 0 | 0 | - | 0 | l | - | m | m |
| Split | - | - | h | d | 0 | f | 0 | n | 0 | - | - | m | m |

-: ignore_event

0: postpone_event

a:send_NewCM_multicast_msg
start_startup_timer
start_Activity_timer
state=Startup

b: not P0: send_NewCM_msg
start_startup_timer
P0: state=Ready

c: P1: stop_startup_timer
state=Ready

d:mark_Active_flag_for_sender_NM

e:send_CheckAlive_msg_to_timedout_NM
   start_CheckAlive_timer
   state=CrashDetect
f: remove_crashed_N_from_system
g: send_AddN_msg_to_selected_NM
h: P1: stop_Split_timer
   send_startNM_msg_to_selected_N
   state=Ready

i: Add_N_to_system
j: send_Split_multicast_msg
   start_Split_timer
   state=Split
k: stop_CheckAlive_timer
   mark_NMActive_flag_for_senderNM
   state=Ready
l: remove_NM_from_system
   send_StartNM_msg_to_selected_N
   state=Ready
m: send_HB_msg
n: not P2: send_Split_multicast_msg
           start_Split_timer
   P2: state=Ready

P0: max_newCM_msg_sent_count reached
P1: Number of NMs sent NMOK_msg = NMcount
P2: max_Split_msg_sent_count reached

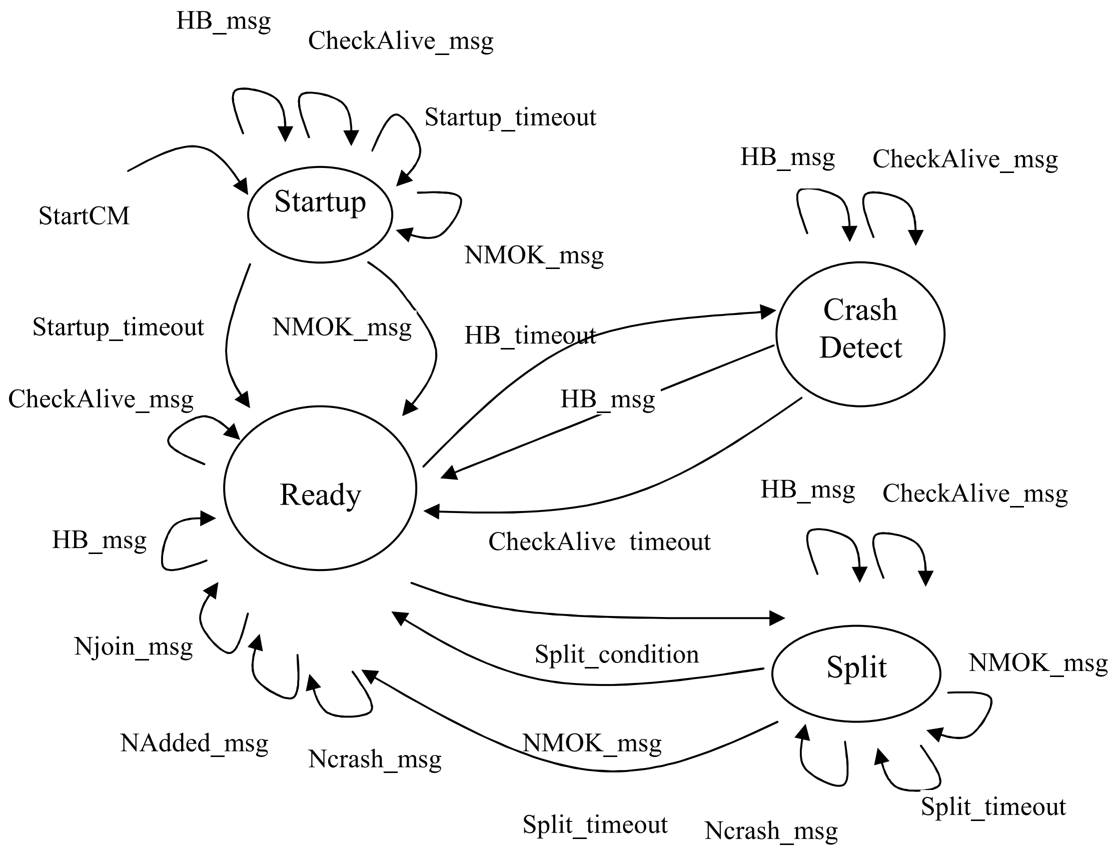


Figure 3.4 State transition diagram of CM module

CM starts initially in a NULL state. First, it sends a NewCM multicast message to inform NMs its awareness, and waits replies with NMOK messages from them in Startup state. It also gets current cluster information by these messages. When all of NMs return replies, CM starts its normal operation in Ready state. When an HB_timeout event occurred which means it is not communicated with an NM in HB_time period, checks NM aliveness by sending a CheckAlive message and waits a HB message in CrashDetect state. CM returns to Ready state if it receives an HB message or CheckAlive timeout occurred. In latter case it decides that the NM is dead and promotes the N that is backup of dead NM as the new NM of that group.

When CM receives a Njoin message, it selects the NM with having the least number of Ns and sends an AddN message containing the address of the new N to that NM. When it receives Nadded message it adds this N to cluster database.

CM turns to Split state when it decides a new NM is needed in the system for scalability issues  because of an increase in the number Ns in the cluster. CM selects the last joined N as the new NM. In this state CM calculates split-count, which is the number of Ns that is to be transferred from the groups to the new group and sends it to NMs with a SPLIT multicast message and starts the new NM. NMs transfer their Ns to the new NM's group. The formula of split-count calculation is :

$$\text{Split count} = \frac{\#\text{ of Ns}}{\#\text{ of NMs} * (\#\text{ of NMs}+1)}$$

CM waits replies to its SPLIT message with NMOK messages from NMs to know the completion of SPLIT operations and returns to Ready state.

The pseudocode of CM module is in Appendix A.

### *3.4.1.2 Node Manager*

**Incoming Events:**

1 StartNM_msg_received
2 Startup_timeout
3 NOK_msg_received
4 HB_msg_received
5 HB_timeout
6 AddN_msg_received
7 Split_msg_received
8 NewCM_msg_received
9 CheckAlive_timeout
10 NewN_timeout
11 NMUpdate_timeout
12 CheckAlive_msg_received
13 Activity_timeout

**Outgoing Events:**

HB_msg_sent
CheckAlive_msg_sent
StartCM_msg_sent
NAdded_msg_sent
NMUpdate_msg_sent
NMOK_msg_sent
NAccepted_msg_sent
NewNM_msg_sent
NCrash_msg_sent

**Event:**

Startup
Ready
NCrashDetect
CMCrashDetect
NewN
Split

Table 3.2 Event-state table of NM module

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NULL | a | - | - | - | - | - | - | - | - | - | - | - | - |
| Startup | - | b | c | d | 0 | 0 | 0 | 0 | - | - | - | p | p |
| Ready | - | - | - | d | e | f | g | h | - | - | - | p | p |
| NCrashDetect | - | - | - | i | 0 | 0 | 0 | 0 | j | - | - | p | p |
| CMCrashDetect | - | - | - | r | 0 | 0 | 0 | 0 | s | - | - | p | p |
| NewN | - | - | k | d | 0 | 0 | 0 | 0 | - | l | - | p | p |
| Split | - | - | m | d | 0 | 0 | 0 | 0 | - | - | n | p | p |

-: ignore_event
0: postpone_event
a: send_NewNM_multicast_msg
start_startup_timer
start_Activity_timer
state=Startup
b: not P0: send_NewNM_msg
start_startup_timer
P0: state=Ready
c: P1: stop_startup_timer
state=Ready
d:mark_NMActiveflagforsenderN/CM
e:send_CheckAlivemsgTotimedoutN/CM
start_CheckAlive_timer

state=NCrashDetect/CMCrashDetect
f: send_NAccepted_msg_to_new_N
start_NewN_timer
state=NewN
g: send_NMUpdate_multicast_msg
start_NMUpdate_timer
state=Split
h: send_NMOK_msg
i: stop_CheckAlive_timer

mark_NMActive_flag_for_sender_N
state=Ready
j: remove_crashed_N
send_NCrash_msg_to_CM
state=Ready

r: stop_CheckAlive_timer
mark_NMActive_flag_for_CM
state=Ready
s: send_startCM_msg
state=Ready
k: stop_NewN_timer
Add_node
send_NAdded_msg_to_CM
state=Ready
l: state=Ready
m: not P2:
mark_NUpdated_Flag_for_sender_N
P2: stop_NMUpdate_timer
send_NMOK_msg_to_CM
State=Ready
n: not P3:
send_NMUpdate_multicast_msg
start_NMUpdate_timer

P3: send_NCrash_msg_to_CM
send_NMOK_msg_to_CM
State=Ready
p: send_HB_msg

P0: max_newNM_msg_sent_count reached
P1: Number of Ns sent NOK_msg = Ncount
P2: Number of Ns sent NOK_msg = SplitCount
P3: max_NMUpdate_msg_sent_count reached

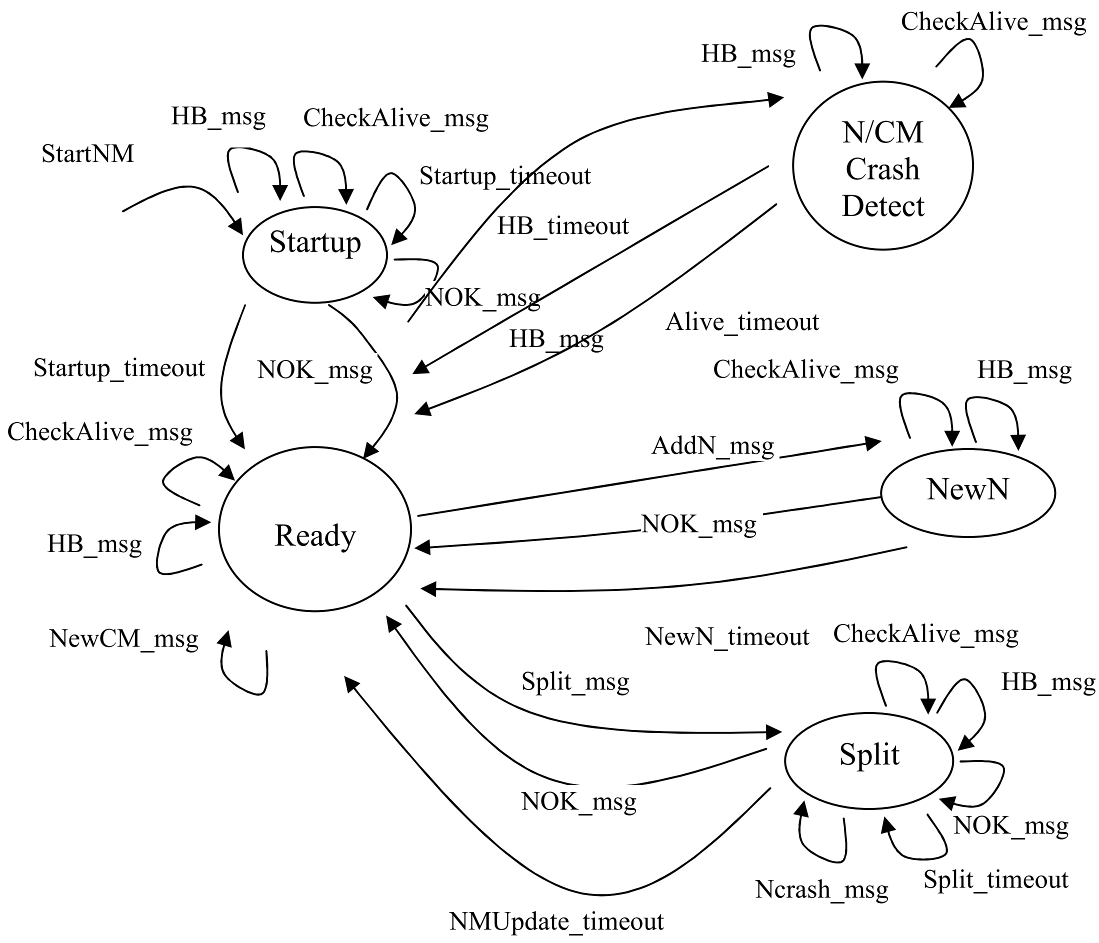


Figure 3.5 State transition diagram of NM module.

When NM starts, it sends NewNM multicast message to its group to inform its Ns its awareness and turns to Startup state. When all Ns relies with a NOK message NM turns to Ready state and starts its normal operation. NM turns to NCrashDetect state if HB_timeout occurred and sends a CheckAlive message. It returns to Ready state if N replies with an HB message received or Alive_timeout occurred. In the latter case it removes this dead N and informs CM that its N is dead with a NCrash message.

If NM is the backup of CM, it can turn to CMCrashDetect state when it is not communicated with CM in HB_time interval and sends a CheckAlive message to CM. If CM does not reply with a HB message in CheckAlive time interval, NM decides that it is dead and promotes itself as the new CM.

If NM receives a AddN message, it joins the new N to its group and sends a NAccepted message to it to inform its awareness and gives necessary data about the group and the cluster and turns to NewN state. When the new N replies with a NOK message, NM sends NAdded message to CM and returns to Ready state.

When NM receives SPLIT message it sends NMUpdate multicast message to its group which contains a number that indicates the Ns to be transferred (calculated as: # of Ns in group - split-count. Ns whose Ids are greater than this number will be transferred). When Ns reply with NOK messages NM sends NMOK message to CM and returns to Ready state.

The pseudocode of NM module is in Appendix A.

### *3.4.1.3 Node*

| **Incoming Events:** | **Outgoing Events:** |
|---|---|
| 1 StartN_msg_received | HB_msg_sent |
| 2 Startup_timeout | Alive_msg_sent |
| 3 NAccepted_msg_received | NJoin_msg_sent |
| 4 CheckAlive_msg_received | NOK_msg_sent |
| 5 NMUpdate_msg_received | |
| 6 NewNM_msg_received | **States:** |
| 7 Activity_timeout | Startup |
| | Ready |

Table 3.3 Event-state table of N module

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| NULL | a | - | - | - | - | - | - |
| Startup | - | b | c | - | - | - | - |
| Ready | - | - | - | d | e | f | d |

-: ignore_event

a: send_NJoin_multicast_msg
start_Startup_timer
State=Startup

b: send_NJoin_multicast_msg
start_Startup_timer

c: stop_Startup_timer
send_NOK_msg
start_Activity_timer
State=Ready

d: send_HB_msg

e: Update_NM_info
Send_NOK_msg_to_old_N

f: Update_NM_info
Send_NOK_msg_to_new_N

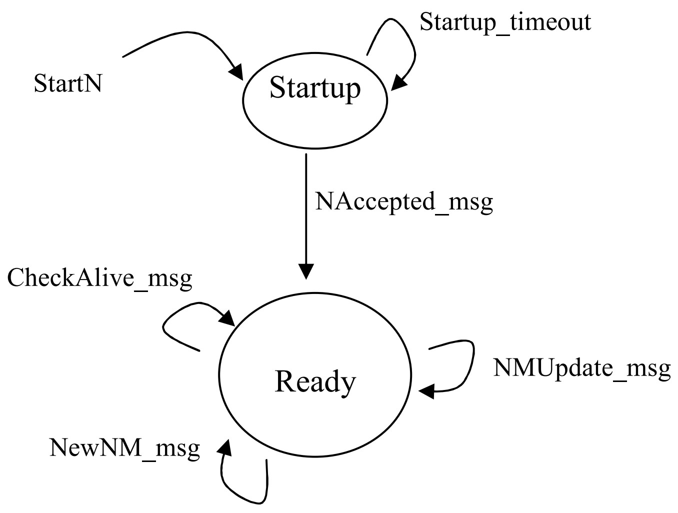


Figure 3.6 State transition diagram of N module.

N represents a worker (processor) and its main responsibility is to run jobs submitted to it. All workstations in the cluster run a N process which means the overhead of N process effects all of the workstations so the whole cluster. As a result N is designed as a simple module to cause as low as possible overhead.

When N starts, it sends a NJoin message to CM and turns to Startup state. It turns to Ready state when NAccepted message is received and starts its normal operation.

When N receives a NMUpdate message, it checks its ID with split-count in the message. If its ID is greater than split-count, it changes its group information with the new data in the message, so the N is transferred to a new group.

The pseudocode of N module is in Appendix A.

### *3.4.2 Message And Time Analysis*

#### *3.4.2.1 Joining A Node*

The flow of messages for the operation of joining a node to the cluster is shown in Table 3.4. The operation is completed by transmission of 5 messages and the time required for this operation is *5t* with *t* representing the time to transmit one message. Note that the operation does not depend on the size of cluster and is not effected by the change in the number of Ns or NMs.

Table 3.4 Message flow for N join operation

| Message | From | To | Time |
|---|---|---|---|
| 1. NJoin | N | CM | t |
| 2. AddN | CM | NM | t |
| 3. NAccepted | NM | N | t |
| 4. NOK | N | NM | t |
| 5. NAdded | NM | CM | t |

#### *3.4.2.2 Node Crash Detection and Removal*

The flow of messages for the operation when a node crash occurs is shown in Table 3.5. The crash detection operation starts with a *Heartbeat timeout* event and stops with *CheckAlive timeout* event. The removal operation is completed by sending the *NCrash* message to CM. The detection and removal operation is completed by the transmission of 2 messages and the duration of the operation is *HB+CA+t*. Note that the operation does not depend on the size of cluster and is not effected by the change in the number of Ns or NMs.

Table 3.5 Message flow for N crash detection and removal operation

| **Message** | **From** | **To** | **Time** | **Event** |
|---|---|---|---|---|
| - | - | - | Heartbeat_time (HB) | HB_timeout occured |
| 1.CheckAlive | NM | N | t | |
| - | - | - | CheckAlive_time (CA) | CheckAlive_timeout occured |
| 2.NCrash | NM | CM | t | |

#### *3.4.2.3 Node Manager Crash Detection and Recovery*

The flow of messages for the operation when a node manager crash occurs is shown in Table 3.6. The crash detection operation starts with a *Heartbeat timeout* event and stops with *CheckAlive timeout* event. The recovery operation starts by the CM with a *StartNM* message and then the new NM sends a multicast *NewNM* message to its group. The operation is completed by the receipt of *NOK* messages from the Ns in the group. message to CM. The number of messages transmitted during this operation is *3+N/NM* where *N/NM* represents the number of Ns in the group of the crashed NM. The time spent for this operation is *HB+CA+3t*. Note that the operation depends on the size of a group and the transmission of the number of messages is effected by the change in the number of Ns in a group.

Table 3.6 Message flow for NM crash detection and recovery operation

| **Message** | **From** | **To** | **Time** | **Event** |
|---|---|---|---|---|
| 1. - | - | - | HB | HB_timeout occured |
| 2.CheckAlive | CM | NM | t | |
| - | - | - | CA | CheckAlive_timeout occured |
| 3. StartNM | CM | NM | t | |
| 4. NewNM | NM | MulticastGroup | t | |
| 5. NOK | Ns in group | NM | t | |

*3.4.2.4 Cluster Manager Crash Detection and Recovery*

The flow of messages for the operation when a cluster manager crash occurs is shown in Table 3.7. The crash detection operation starts with a *Heartbeat timeout* event and stops with *CheckAlive timeout* event. The recovery operation starts by the promotion of backup NM as the new CM and then the new CM sends a multicast *NewCM* message to NMs in the cluster. The operation is completed by the receipt of *NMOK* messages from the NMs to the CM. The number of messages transmitted during this operation is *2+NM* where *NM* represents the number of NMs in the cluster. The time spent for this operation is *HB+CA+2t.* Note that the operation depends on the size of the cluster and the transmission of the number of messages is effected by the change in the number of NMs in the cluster.

Table 3.7 Message flow for CM crash detection and recovery operation

| **Message** | **From** | **To** | **Time** | **Event** |
|---|---|---|---|---|
| 1. - | - | - | HB | HB_timeout occured |
| 2.CheckAlive | NM | CM | t | |
| - | - | - | CA | CheckAlive_timeout occured |
| 3. - | - | - | - | StartCM |
| 4. NewCM | CM | MulticastCluster | t | |
| 5. NMOK | NMs in cluster | CM | t | |

### *3.4.2.5 Cluster Split and Starting a New NM*

The flow of messages for the operation for a split operation is shown in Table 3.8. Split operation is started by the CM by sending a *Split* multicast message. Upon receiving this message NMs send multicast *NMUpdate* messages to their groups. Then CM starts the new NM by sending a *StartNM* message to the new NM. After that the new NM sends a multicast *NewNM* message to its new group. The operation ends by the *NOK* messages from Ns of the new group. The number of messages sent during the operation and spent time is :

*Split operation:*
Messages : *1+NM+NM*s*
s: split-count = *N/(NM*(NM+1))*
Time : *4t*

*Start of New NM :*
Messages : *2+NM*s*
Time : *3t*

*Total:*
Messages: *3+NM*(2*s+1)*
Time: *6t*

The operation depends on the size of the cluster and the transmission of the number of messages is effected by the change in the number of NMs and Ns in the cluster.

Table 3.8 Message flow for a split operation

| **Message** | **From** | **To** | **Time** |
|---|---|---|---|
| 1.Split | CM | MulticastCluster | t |
| 2.NMUpdate | NM | MulticastGroup | t |
| 3.NOK | Ns | NM | t |
| 4.NMOK | NMs | CM | t |
| 5. StartNM | CM | N | t |
| 6. NewNM | NM | MulticastGroup | t |
| 7. NOK | Ns in group | NM | t |

# CHAPTER FOUR
# THE LOAD BALANCING MODEL

## 4.1 System Architecture

The load balancing model is placed on top of the CIM architecture. CIM is designed as a hierarchically centralized model containing three components. On the top there is a Cluster Manager (CM) which constructs the cluster and manages the resources and components of the whole cluster. It does its job via a number of low level managers called Node Managers. A Node Manager (NM) is responsible for managing a discrete subset of the nodes of the cluster. A Node (N) represents a single processing unit, a worker of the cluster, such as an independent workstation. CM assigns each joined N to a an NM. To manage Ns in a scalable manner, CM determines the number of NMs according to the number of Ns. As number of Ns increases CM employs new NMs. So, the Ns are grouped by a number of NMs. Each group is called by its NM (e.g., NM group 1, NM group 2, etc.)

The load balancing model inherits the CIM architecture and its components. Each N has its own resources like memory, CPU, disk, etc., so the load balancing model should distribute the workload among the Ns of the cluster. To balance the workload in a scalable manner, the hierarchically layered architecture is used as local and global load sharing concepts as shown in Figure 4.1.

In local load sharing, the distribution of workload is performed among the group of Ns of an NM group. This type of distribution is called "local" because it involves a subset of the Ns belonging to the same NM. Local load sharing can also be called as partial load balancing, as it distributes the partial workload (workload in an NM group) of the whole cluster. Local load sharing can be performed in more than one NM group in parallel at the same time. NMs are responsible for running the local load sharing scheme in their groups.

As its name reminds, global load sharing scheme aims to distribute the workload of the whole cluster. In global load sharing, the system tries to balance workload between NM groups, so load distribution is performed on Ns belonging to different NM groups. CM is responsible for determining the need for the global load sharing, and then tell NMs to run it.

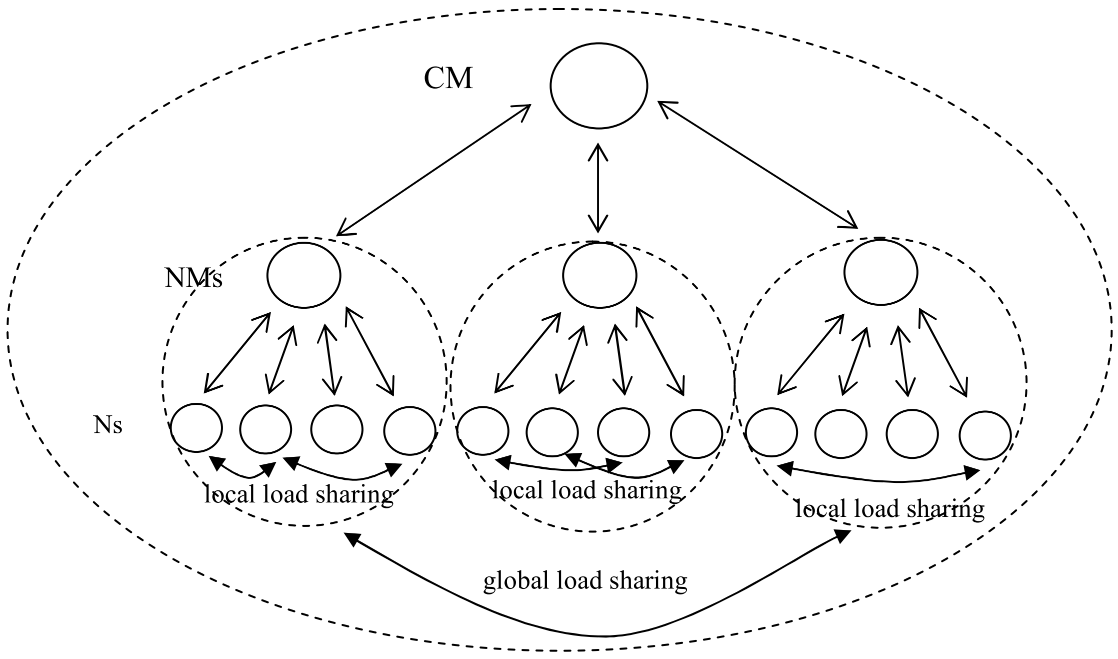


Figure 4.1 Hierarchical architecture of  load balancing model.

## 4.2 Messaging Infrastructure

Communication architecture of the load balancing model is based on the CIM messaging infrastructure. Group messaging is based on multicasting. Each NM has a multicast group containing their Ns. CM also has a multicast group that has all NMs as members. Multicast group structure is shown in Figure 4.2. Point to point messaging is also allowed when necessary, such as in load transfers.

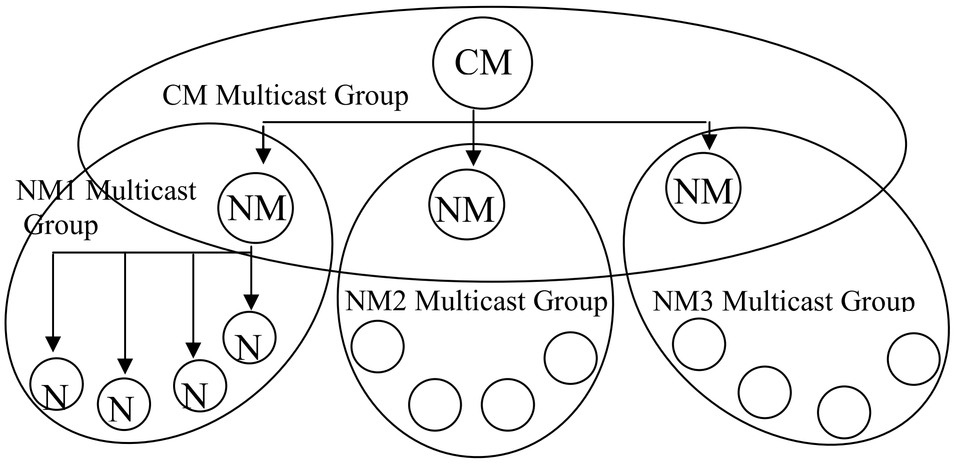


Figure 4.2 Multicast groups in load balancing model.

## 4.3 Load Balancing Algorithm

As discussed in section 3.1, the load balancing scheme hierarchically consists of two parts: local and global load sharing. The algorithms for these parts are described in terms of load balancing algorithm concepts as mentioned in section 2. Both parts are in the class of adaptive load balancing algorithms.

### *4.3.1 Local Load Sharing*

In local load sharing, each NMs try to distribute workload of their groups among their Ns. To determine whether load sharing is needed, NM collects load information from its Ns. Using these information NM calculates some adaptive threshold values. Using these values NM determines whether load distribution is needed and which Ns are involved in load transfer. Load is transferred from heavily loaded Ns to lightly loaded Ns.

**Transfer Policy:** Based on its knowledge about the load states of its Ns, NM calculates threshold values based on average load of its group and determines if a balancing is needed.

**Selection Policy :** Task selection for transfer is a complicated task. To avoid high overheads, firstly jobs that has not been started yet are selected to transfer. This type of load transfers are called non-preemptive. If more fine-grade balancing is desired tasks that are in execution can be selected for migration which is called preemptive task transfers.

**Location Policy :** By using calculated threshold values, NM selects a pair of Ns to start task transfer. Ns that have load values greater than upper threshold are selected as sender, ones that fall below lower threshold become receivers for task transfers. Task transfer is then performed from sender to receiver Ns.

**Information Policy :** The information needed for the algorithm is the local load values of Ns. In this model, Ns periodically report their load states to their NM. NM uses these information in other policies of load balancing for making decisions.

### *4.3.2 Global Load Sharing*

While local load sharing scheme distributes workload of group of Ns, global load sharing tries to share the load between groups. Thus global load sharing involves sharing of load between Ns that are in different groups. Global load sharing scheme completes partially balanced state of local load sharing groups to a globally shared state of the whole system. In this scheme, CM collects load state information of NM groups from NMs. With these information, CM determines whether the global load sharing is needed and if so informs NMs to start it. NM of sender and receiver groups, select suitable Ns and start task transfer.

**Transfer Policy:** Based on its knowledge about the load states of NM groups, CM calculates threshold values based on average load of the whole system and determines if a balancing is needed.

**Selection Policy :** As in local load sharing, to avoid high overheads, non-preemptive task transfers are preferred. Preemptive transfers provide better balancing ratios but cause much more overhead to the system.

**Location Policy :** With calculated threshold values, CM determines sender and receiver NMs and informs all NMs about its decisions. Then a sender NM selects a receiver NM and requests a task transfer. After receiver NM accepts the request, both NMs select a pair of Ns for task transfer. Ns that have load values greater than upper threshold is selected as sender, ones that fall below lower threshold become receivers for task transfers. Task transfer is then performed from sender to receiver Ns.

**Information Policy :** The information needed for the global load sharing algorithm is the average load values of NM groups. In this model, NMs periodically report the load states of their groups to CM. CM uses these information in other policies of load balancing for decision making.

### *4.3.3 Load Information*

The most important input of the load balancing algorithm is the load information. So, it is important to collect the correct load values from the Ns of the system. To determine the current load value of a N, one or more load indices can be calculated. Typical load indices are CPU load average, memory usage, I/O queue length, network bandwidth utilization, etc.

The load balancing model is designed for using multiple load indices at the same time as desired. By using configured load indices, each N calculates its load value ($L$). This single load value is used as N's current load state and reported to NM. To compute the load value, each load index ($l_i$) is given a percentage weight ($w_i$). The weight of a load index specifies its degree of importance effect over the load value. The load value is computed as;

$$L = \sum_{i=1}^{n} l_i w_i$$

Also, each load index has a threshold value ($t_i$). If the value of a load index reaches or exceeds its threshold, then it directly effects the computed load value regardless of its weight and weights of other indices are reduced. This protects the usage of a single resource exceeding its capacity while others are low. In this case the load value is computed as;

$$L = l_i + \sum_{k \neq i}^{n} l_k \overline{w}_k \quad , \quad \overline{w}_k = \frac{w_k}{100 - w_i}(100 - l_i) \quad \text{where} \quad l_i > t_i \, .$$

If more than one index reaches its threshold, then the most important one is taken as threshold exceeded index.

For example consider a system in which load balancing scheme is configured having load indices as average CPU usage and memory usage with equal weights. At a certain time when the CPU usage of a N is as high as 90% while memory usage is very low such as 20%, the computed load index gives a moderate load value of 55%. But this N should be considered as highly loaded, since its CPU capacity is almost consumed totally. Configuring the CPU load index having a threshold value of 80%, as the value 90% exceeds the threshold it directly effects the load value computation, so N's load value becomes 92% (90 for CPU usage + 10% of 20 for memory usage, since its weight is reduced to 10%).

### *4.3.4 Load Sharing Thresholds*

In transfer policy and location policy, the load balancing algorithm determines whether a balancing operation is needed and which parts will be involved in load transfers as senders and receivers. Adaptive load sharing thresholds are used for this purpose. They are adaptive, since their values are regulated according to the load level of the system. For example, the load of the system is high when the most of the

Ns are heavily loaded, and the thresholds are increased to prevent useless task transfers. Also, in case of a very low load levels, system does not need load sharing, so thresholds are be set to appropriate values to protect the system from ineffective task transfers. In that mean, adaptivity of load sharing thresholds are sensitivity of the load balancing algorithm.

The calculation of the load thresholds is a customizable task. It can be changed, its sensitivity can be customized according to the needs and expectations from the load balancing algorithm. In this project, two threshold values named sender threshold ($T_S$) and receiver threshold ($T_R$) to select senders and receivers. Members (Ns or NM groups) with load values exceeding $T_S$ are selected as senders and members having lower load values than $T_R$ are treated as receivers. Thresholds computations are based on average load values:

$$\text{Average Load: } L_A = \frac{1}{n}\sum_{i=1}^{n} L_i$$

$$T_R = L_A - \frac{1}{r}\sum_{i=1}^{n}(L_A - L_i) \quad \text{for all } L_i < L_A\text{, } r = \#\text{ of members where } L_i < L_A$$

$$T_S = L_A + \frac{1}{s}\sum_{i=1}^{n}(L_A - L_i) \quad \text{for all } L_i < L_A\text{, } s = \#\text{ of members where } L_i > L_A$$

## 4.4 Formal Protocol Design of Load Balancing Model

The protocol of load balancing model is designed using finite state machines (FSM). In a FSM; states, incoming events and outgoing events are defined. Then operations performed in occurance of incoming events at related states are described in an event-state table. After that the state transition diagram is generated which shows state changes according to occurance of incoming events. Finally pseudocodes are generated by using designed FSMs.

Based on the CIM, CM, NM and N components are designed as different modules in load balancing model too, so their FSMs are designed separately. In implementation these modules can be separate processes or threads.

### *4.4.1 Finite State Machines*

#### *4.4.1.1 Cluster Manager*

**Incoming Events:**

1 StartCM_msg_received

2 NMLoadInfo_msg_received

3 GlobalBalancing_condition_occurred

**Outgoing Events:**

GlobalLoadInfo_msg_sent

**States:**

Ready

Table 4.1 Event-state table of CM module

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| NULL | a | - | - | - |
| Ready | - | b | c | d |

-: ignore_event

a: Initialize_info
   state=Ready

b: Update_global_load_info
   Check_GlobalBalancing

c: Calculate_thresholds & Specify_Receivers
   Send_GlobalLoad_Info_multicasticast_Msg

d: Update_global_load_info

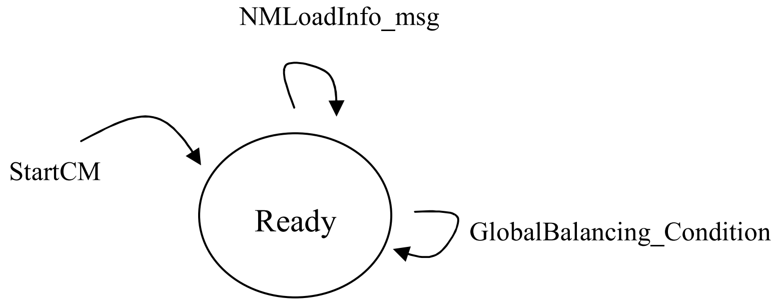


Figure 4.3 State transition diagram of CM module.

When CM starts with a StartCM message in CIM protocol, the load balancing module of CM also starts which has a single-state FSM. CM, collects load values of NM groups by NMLoadInfo messages sent by NMs. CM calculates sender and receiver threshold values to classify NM groups, and when it decides a global load sharing is needed, it multicasts a GlobalLoadInfo message to NMs to start global load sharing. Threshold values and receiver NM addresses are contained in that message.

CM waits replies to its SPLIT message with NMOK messages from NMs to know the completion of SPLIT operations and returns to Ready state.

When CM starts with a StartCM message, the load balancing module of CM also starts. CM, collects load values of NM groups by NMLoadInfo messages sent by NMs. CM calculates sender and receiver threshold values to classify NM groups, and when it decides a global load sharing is needed, it multicasts a GlobalLoadInfo message to NMs to start global load sharing. Threshold values and receiver NM addresses are contained in that message.

The flowchart of CM module is in Appendix B.

*4.4.1.2 Node Manager*

**Incoming Events:**

1 StartNM_msg_received
2 NLoadInfo_msg_received
3 LocalBalancing_condition_occurred
4 GlobalLoadInfo_msg_received
5 GlobalLoadTransfer_Request_msg_received
6 GlobalLoadTransfer_Accepted_msg_received
7 GlobalLoadTransfer_Rejected_msg_received
8 GlobalLoadTransfer_Timeout
9 Activity_timeout
10 NCrashMsgReceived

**Outgoing Events:**

TransferLoad_msg_sent
GlobalLoadTransfer_Request_msg_sent
GlobalLoadTransfer_Accepted_msg_sent
GlobalLoadTransfer_Rejected_msg_sent
NMLoadInfo_msg_sent

**States:**

Ready
GlobalLoadTransfer

Table 4.2 Event-state table of NM module

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| NULL | a | - | - | - | - | - | - | - | - | - |
| Ready | - | b | d | e | g | - | - | - | k | l |
| GlobalLoadTransfer | - | c | - | f | h | i | j | j | k | l |

```
-: ignore_event
a: Initialize_info
   state=Ready
b: Update_local_load_info
   Check_LocalBalancing
c: Update_local_load_info
d: Calculate_thresholds & Specify_Sender&Receivers
   Send_TransferLoad_msg_to_senderN
e: Update_GlobalLoadInfo
   P0: Take_a_receiverNM_from_ReveiverNMs_list
       Send_GlobalLoadTransfer_Request_msg_to_receiverNM
       Start_GlobalLoadTransfer_timer
       state=GlobalLoadTransfer
f: Update_GlobalLoadInfo
g: not P1: Send_GlobalLoad_transfer_Reject_msg_to_senderNM
   P1: Choose_suitable_receiverN
       Send_GlobalLoadTransfer_Accepted_msg_to_senderNM
h: Send_GlobalLoad_transfer_Reject_msg_to_senderNM
i: Stop_GlobalLoadTransfer_timer
   Choose_suitable_SenderN
   Send_TransferLoad_msg_toSenderN
   state=Ready
j:  Stop_GlobalLoadTransfer_timer
   not P2: Send_GlobalLoadTransfer_Request_msg_to_receiverNM
       Start_GlobalLoadTransfer_timer
```

P2: state=Ready

k: send_NMloadInfo_msg_to_CM

l: Update_local_load_info

P0: Global_SenderLoadThreshold_exceeded

P1: Global_ReceiverLoadThreshold_exceeded

P2: End_of_ReceiverNMs_List

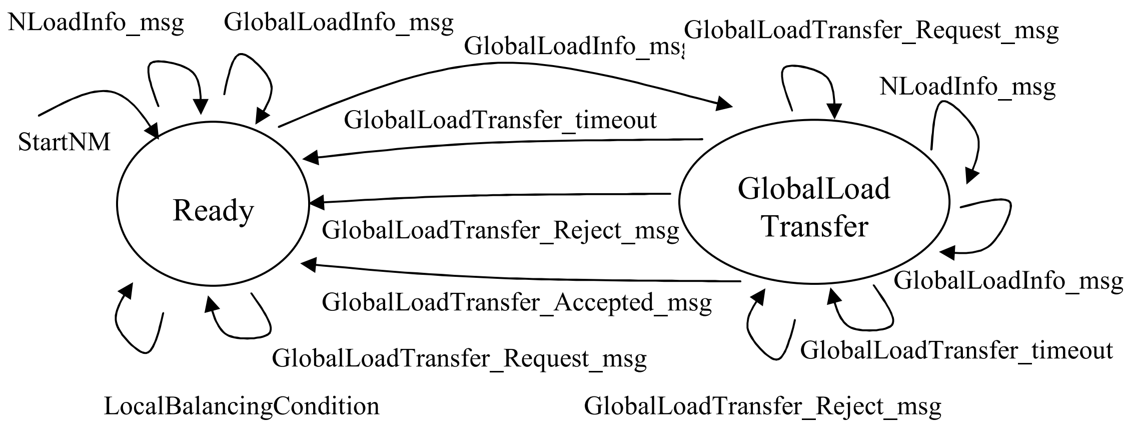

NLoadInfo_msg
GlobalLoadInfo_msg
GlobalLoadInfo_msg
GlobalLoadTransfer_Request_msg
NLoadInfo_msg
StartNM
GlobalLoadTransfer_timeout
Ready
GlobalLoad Transfer
GlobalLoadTransfer_Reject_msg
GlobalLoadInfo_msg
GlobalLoadTransfer_Accepted_msg
GlobalLoadTransfer_timeout
GlobalLoadTransfer_Request_msg
LocalBalancingCondition
GlobalLoadTransfer_Reject_msg

Figure 4.4 State transition diagram of NM module.

When NM starts with a StartNM message in CIM protocol, the load balancing module of NM also starts. NM collects load states of its Ns by NloadInfo messages. It calculates average load of its group, and also the threshold values to classify sender and receiver Ns. If it decides to run local load sharing, NM sends a TransferLoad message to sender N. This message contains the receiver N address.

NM reports load state of its group to CM by putting it in HeartBeat messages that are sent to inform its aliveness in CIM protocol. When NM receives a GlobalLoadInfo message, it compares global load sharing threshold values with its load state and realize that whether it is a sender, sends a GlobalLoadTransfer_Request message to a receiver NM address selected from the receivers list contained in the GlobalLoadInfo message. If that transfer request is unsuccessful (if request is timed out or rejected), it continues load requests by selecting another receiver NM address from the receivers list. If request is accepted, it sends a TransferLoad message to a selected sender N address to start global load transfer.

Upon receiving a GlobalLoadTransfer_Request message, NM checks to see if it is a receiver by comparing its current load state with global load sharing threshold values, and if so selects a N for receiving load and sends its address to sender NM via a Global_loadTransfer_Accepted message.

The flowchart of NM module is in Appendix B.

### *4.4.1.3 Node*

**Incoming Events:**

1 StartN_msg_received
2 TransferLoad_msg_received
3 LoadTransfer_msg_received
4 LoadTransfer_OK_msg_received
5 LoadTransfer_timeout
6 Activity_timeout

**Outgoing Events:**

LoadTransfer_msg_sent
LoadTransfer_OK_msg_sent
LoadTransfer_timeout
NloadInfo_msg_sent

**States:**

Ready
LoadTransfer

Table 4.3 Event-state Table of N module

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| NULL | a | - | - | - | - | - |
| Ready | - | b | c | - | - | f |
| LoadTransfer | - | - | - | d | e | f |

-: ignore_event
a: Initialize_info
state=Ready
b: select_task_to_transfer
send_LoadTransfer_msg_to_receiverN
send_NloadInfo_msg_to_NM

d: stop_LoadTransfer_timer
state=Ready
e: state=Ready
f: calculate_local_load_value

```
    start_LoadTransfer_timer
    state=LoadTransfer
c: get_task_info
    send_LoadTransfer_OK_msg_to_senderN
```

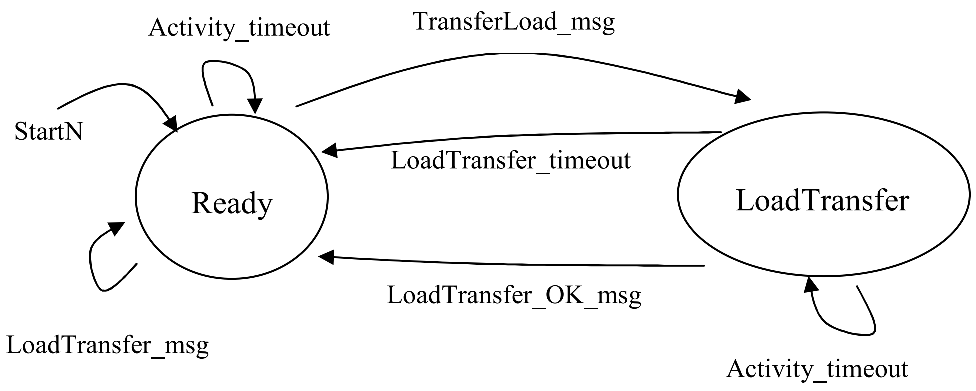


Figure 4.5 State transition diagram of N module.

When N starts with a StartN message in CIM protocol, the load balancing module of N also starts. N reports its load state to CM by putting it in HeartBeat messages that are sent to inform its aliveness in CIM protocol. When it receives a TransferLoad message, selects a task to transfer and sends a LoadTransfer message to receiver N contained in TransferLoad message. If N receives a LoadTransfer message, its gets the task information and informs sender N about the success of the task transfer by sending LoadTransfer_OK message.

The flowchart of N module is in Appendix B.

### *4.4.2 Message and Time Analysis*

#### *4.4.2.1 Local Load Sharing*

The flow of messages for a local task transfer operation is shown in Table 4.4. Operation starts with a *TransferLoad* message from NM to the sender N. Then sender N sends a *LoadTransfer* message to the receiver N. The operation ends with a *LoadTransferOK* message. Hence the operation is completed with 3 messages in *3t* unit time were *t* is the time spent to transfer a message. This operation is not effected by the size of the cluster.

Table 4.4 Message flow of a local load transfer operation

| Message | From | To | Time |
|---|---|---|---|
| 1. TransferLoad | NM | N | t |
| 2. LoadTransfer | N (sender) | N (receiver) | t |
| 3. LoadTransfer_OK | N (receiver) | N (sender) | t |

### *4.4.2.2 Global Load Sharing*

The flow of messages for a global task transfer operation is shown in Table 4.5. Operation starts with a *GlobalLoadInfo* multicast message from CM to NMs. Then a sender NM sends a *GlobalLoadTransfer_Request* message to a receiver NM. Upon receiving a reply with a *GlobalLoadTransfer_Accepted* message, the sender NM sends *TransferLoad* message to the sender N. The operation is completed by the sender and receiver Ns ends with the transmission of *TransferLoad* and *LoadTransferOK* messages. During a global load transfer operation is completed with 6 messages in *6t* unit time were *t* is the time spent to transfer a message. Hence the messaging in this operation is not effected by the size of the cluster.

Table 4.5 Message flow of a local load transfer operation

| Message | From | To | Time |
|---|---|---|---|
| 1. GlobalLoadInfo | CM | NMs | t |
| 2. GlobalLoadTransfer_Request | NM | NM | t |
| 3. GlobalLoadTransfer_Accepted | NM | NM | t |
| 4. TransferLoad | NM | N | t |
| 5. LoadTransfer | N | N | t |
| 6. LoadTransfer_OK | N | N | t |

# CHAPTER FIVE
# THE IMPLEMENTATION

## 5.1 The Implementation of CIM

### *5.1.1 Multithreaded Process Architecture*

In CIM, each component (CM, NM, and N) is designed as a separate autonomous module. This autonomy is implemented by developing a multithreaded architecture. A single CIM process runs on each workstation of the cluster. On the workstation that performs CM role, runs the CM module thread within the CIM process. Similarly, each NM workstation runs a NM module thread, and workstations running N threads are the nodes of the cluster.

Each module also contains helper threads. A messenger thread receives a message from the message queue and feeds the module thread for processing. Another helper thread, called multicast receiver listens the multicast group port, receives and then puts multicast messages to the module thread's message queue. Similarly, the unicast receiver thread is responsible for listening the unicast port and receiving point-to-point messages. There is also a sender thread that is responsible for packing and sending both unicast and multicast messages to the network. These helper threads are stateless and blocking threads. This means they simply sleep waiting for a message (on a network port or a queue), wakeup when available, serve and then sleep again. These helpers prevent blocking of the module thread. By this implementation the module thread is free for running, such as processing events, performing its internal operations, preparing messages for sending and so on.

The main thread of the CIM process is responsible for maintaining the module threads, such as starting and stopping them when necessary. The multithreaded topology of the CIM implementation is shown in Figure 5.1.

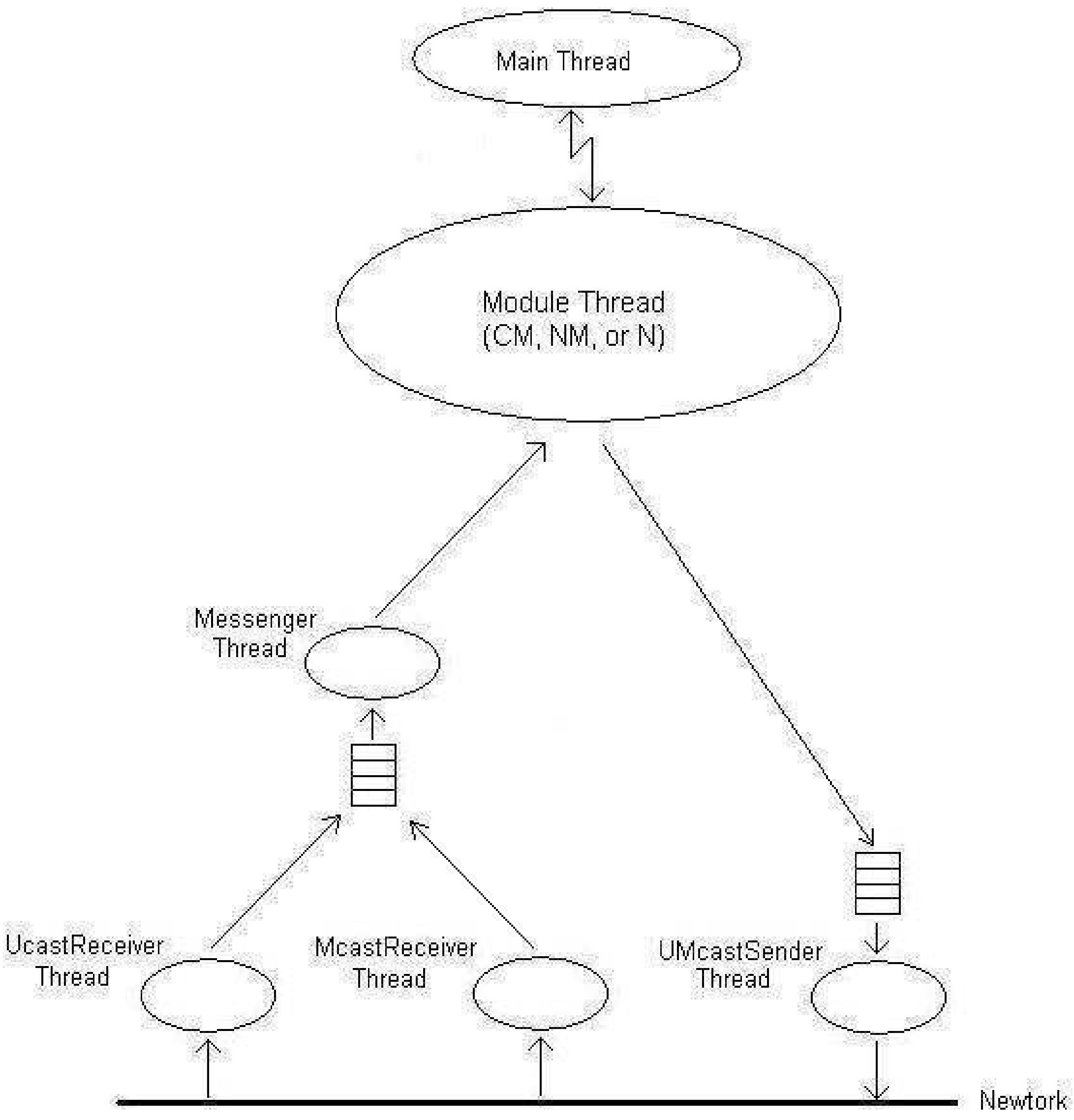


Figure 5.1 Multithreaded architecture of CIM process.

The protocol of CIM was designed formally with finite state machines (FSMs). States, events, event-state tables and state transition diagrams describe the operation of a module. Module implementations are performed using these FSMs. Thus module threads are stateful and event-based. The module is always in a specific state, and when an event occurs, it performs predefined operations on that state, and transits to another state according to the results of the operation, or stays in the current state. An event can be an arrived unicast or multicast message, a timeout occurance, or a result of an internal function running periodically or on a specific situation. All these events, states, operations and state transitions were defined in FSM.

### *5.1.2 Thread Implementation, Communication, Synchronization and Timers*

The implementation of CIM protocol is developed on Linux operating system using standard (GNU) C programming language. Threads are implemented using Posix Threads (pthreads) library which is also available in Linux environment.

Since threads are in a shared memory environment, which means threads belonging to the same process can reach the whole process memory scope, there is no need an IPC mechanism (PIPEs, etc.) which is time-consuming operations causing overhead. For intra-thread communication, FIFO (first in first out) queue structures are designed as shown in Figure 5.2. Every module thread has an input message queue that is controlled by messenger helper threads. Received messages are put those queues by receiver helper threads, and read by messenger thread and fed to the module one by one. Message sender thread also has a local input queue. Messages to be sent are put on this queue by a module thread, then read and sent by the sender thread.

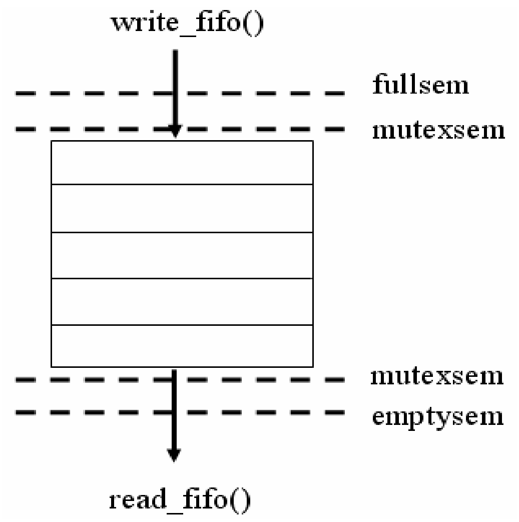


Figure 5.2 FIFO queue structure.

Since these FIFO queue structures can be naturally reachable by more than one thread at the same time, a synchronization and critical section access mechanism should be employed to prevent conflicts. For this purpose a semaphore structure is designed and implemented. Although standard POSIX semaphore library can be used, it is generally designed for inter-process synchronization, and for thread synchronization condition variable structures available in posix thread library is recommended for performance and reliability. A semaphore mechanism is designed using these condition variables as shown in Figure 5.3. Each FIFO queue is protected by semaphores. Besides, threads can be blocked on an empty queue for a message ready to be read, or on a full queue for an available space to be written. Semaphores are also used for other thread synchronization purposes, such as blocking a thread on a situation and waiting for another thread completing its operation.

```
int semaphore_down (Semaphore * s)
{
 pthread_mutex_lock(&(s->mutex));
if (s->value<1) pthread_cond_wait(&(s->cond),&(s->mutex));
   s->value--;
 pthread_mutex_unlock(&(s->mutex));
  return (1);
}

int semaphore_up (Semaphore * s)
{
 pthread_mutex_lock(&(s->mutex));
 s->value++;
 if (s->value<=1) pthread_cond_signal(&(s->cond));
 pthread_mutex_unlock(&(s->mutex));
  return (1);
}
```

Figure 5.3 Semaphore structures.

Timers are also important in protocol design. To generate some timeout events a timer mechanism is used. Timer mechanism should be in millisecond granularity. One solution is using a system interrupt called SIGALRM. When set, process is interrupted and a defined subroutine is called at timeout situation by the system, then the process continues where it is interrupted. With this mechanism a millisecond based timer structure is implemented, and by this structure multiple timers can be set at the same time by the threads of the process.

### *5.1.3 Implementation of Communication*

Network communication in CIM protocol is based on Internet Protocol (IP). The connectionless transport protocol UDP is used for point-to-point communication, and IP Multicast is used for multicast communication. Considering performance and low latencies, in reliable network environments connectionless protocols are suitable for transmitting small stand-alone messages in asynchronous communication.

Network communication operations are kept out of the module threads and performed by helper threads. Since these are blocking operations this separation brings freedom to operations of modules while providing implementation modularity and simplicity. UcastReceiver thread is responsible for receiving unicast messages and forwards them to destination modules by putting them into their FIFO queues. For this purpose, the thread binds, blocks and listens a specific UDP port. Similarly, McastReceiver thread receives incoming multicast messages by binding to a specified multicast group IP address, and listening a specific port. Message sending operations are performed by UMcastSender helper thread. A module that has a unicast or multicast message to be sent puts the message into the FIFO queue of UMcastSender and continues its operation. UMcastSender takes the message from the queue and sends them in a unicast or multicast IP packet to the specified destination. Standard BSD socket library available in Linux environment was used for network communication implementation.

A message is transferred in a standardized message structure. The message structure is shown in Figure 4. Source and destination address is a specific address structure which contains IP address and module ID that defines the module (each CM,NM or N modules in the system have an assigned globally unique module ID). Module type specifies the destination module as CM, NM or N. Retransmit counter is used for specifying a retransmitted message in case of a failure and prevents processing of duplicate messages. Message type defines the message contained in the data section.

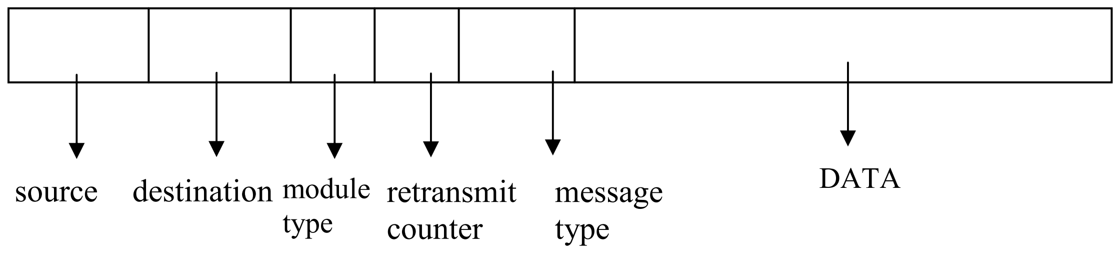


Figure 5.2 Message structure in CIM protocol.

Since messages can contain different data types and structures, each message type has a predefined structure. Hence the data section of a message is interpreted by using message type field. As an example, the structure of message type "NACCEPTED" that is sent by an NM to N as an acceptance to the cluster can be:

```
typedef struct st_naccepted
{
 _address mcastaddr;
 unsigned int mid;
} m_naccepted;
```

When sending this type of message, it is first put into the data section of standard message structure and then sent. Upon receiving, the message type is read and data is put into the "m_naccepted" message structure.

## 5.2 The Implementation of Load Balancing Model

As CIM, the protocol of load balancing model was designed using FSMs too. The implementation of the load balancing model was integrated to the implementation of CIM. CM, NM and N components are designed as different modules in load balancing model. However in implementation, these components are embedded in CIM modules. Hence the same multithreaded architecture as depicted in Figure 5.1

was not changed. The only addition to the implementation is that the module threads in CIM also run CM, NM a N modules of the load balancing module.

# CHAPTER SIX
# EXPERIMENTS AND EVALUATION OF RESULTS

## 6.1 Tests For CIM

The program code of CIM protocol was written with C language for Linux environment and compiled on a kernel version 2.6.3 Mandrake Linux 10.0 (x86) operating system. Since there are not sufficient resources yet for construction of a real cluster environment (tens or even hundreds of workstations are needed), a single machine was used in simulation. For simulation, multiple Ns, NMs and a CM module thread virtually representing different workstations, run on this machine in a single CIM process. The machine was a PC with Intel Celeron 1.7 Ghz processor and 256MB memory.

Tests that will show performance and scalability properties are performed. The performed tests are; joining N to the cluster, failure of N, NM and CM, employing a new NM (split operation). Clusters of different sizes (different number of Ns and/or NMs) were tested. Number of NMs vary from 2 to 16, and total number of Ns vary from 8 to 256 while number of Ns in a NM group vary from 4 to 32.

### *6.1.1 Joining A N To The Cluster*

In this test, time requirements for N join request and acceptance sequence is measured. The join operation starts with the NJOIN message of N. CM gets the message and forwards it to a NM that it selects. Upon receiving NM sends a NACCEPTED message to N. N responds this message with a NOK message, and at last NM informs CM about the completion of the operation with a NADDED message. The number of messages sent during the join operation is constantly 5. Test results of this operation are in Table 6.1. Measured values are in millisecond. Tests show that N join times are around 20 msec and are not effected by the size and shape of the cluster.

Table 6.1 Results of N join test

| | | # of Ns in a NM group | | | |
|---|---|---|---|---|---|
| | | 4 | 8 | 16 | 32 |
| # of NMs | 2 | 22 | 19 | 17 | 19 |
| | 4 | 18 | 17 | 23 | 23 |
| | 6 | | 16 | 17 | 20 |
| | 8 | | 21 | 21 | 25 |
| | 16 | | 20 | 27 | |

### *6.1.2 N Crash Test*

In this test, a randomly selected N thread is stopped instantly, and the operation taken by the cluster is watched. In normal operation, N sends a Heartbeat message when it did not communicate with NM in a certain time interval (Activity time) to inform its aliveness. If N dies, then it will not send Heartbeat messages and NM realizes this situation and sends a CHECKALIVE message to that N and waits an immediate reply within a CheckAlive time period. Since N died, when this period is over, NM removes that N from the cluster and informs CM about this operation. The number of messages sent during this operation is constant and 2. The time measurements are shown in Table 6.2. In table CheckAlive time period is shown as CA. Tests show that the detection and removal operation is around CheckAlive time period and is not effected by the size and shape of the cluster.

Table 6.2 Results of N crash test

| | | # of Ns in a NM group | | | |
|---|---|---|---|---|---|
| | | 4 | 8 | 16 | 32 |
| # of NMs | 2 | CA+0 | CA+0 | CA+1 | CA+1 |
| | 4 | CA+0 | CA+0 | CA+0 | CA+3 |
| | 6 | | CA+0 | CA+3 | CA+2 |
| | 8 | | CA+0 | CA+0 | CA+0 |
| | 16 | | CA+2 | CA+2 | |

### *6.1.3 NM Crash Test*

In this test, a randomly selected NM thread is stopped instantly, and the operation taken by the cluster is watched. In normal operation, NM sends a Heartbeat message if it didn't not communicate with CM in a certain time interval (Activity time) to inform its aliveness. If NM dies, then it will not send Heartbeat messages and CM realizes this situation and sends a CHECKALIVE message to that NM and waits an immediate reply within a CheckAlive time period. Since NM died, when this period is over, CM removes that NM from the cluster and promotes the backup of died NM as the new NM of that group. When the new NM wakes up, it sends a NEWNM multicast message to its group. Upon receiving this message Ns of that group responds it with NOK messages. The number of messages sent during the detection and promotion of new NM is constant and 2. During the start of the new NM, single multicast message is sent, but number of unicast messages depends on the number of Ns in a group.

The time measurements are shown in Table 6.3, and amount of sent messages in Table 6.4. The first sub columns below the labels are for detection and removal, the second ones are for start of the new NM. In tables CheckAlive time period is shown as CA, and volumes of sent multicast messages are shown with an "m" at the end. Tests show that the detection and promotion operation is around CheckAlive time period and is not effected by the size and shape of the cluster. The time required for start of the new NM mainly depends on the size of the group. The time changes by the size of the cluster are ignored since it is caused by performing the tests in a single machine.

Table 6.3 Results of NM crash test

| | | **# of Ns in a NM group** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **4** | | **8** | | **16** | | **32** | |
| **# of NMs** | **2** | CA+2 | 46 | CA+2 | 70 | CA+2 | 96 | CA+2 | 134 |
| | **4** | CA+1 | 52 | CA+1 | 83 | CA+1 | 105 | CA+0 | 145 |
| | **6** | | | CA+2 | 83 | CA+2 | 110 | CA+0 | 169 |
| | **8** | | | CA+1 | 85 | CA+1 | 100 | CA+4 | 170 |
| | **16** | | | CA+0 | 96 | CA+2 | 100 | | |

Table 6.4 Volumes of transmitted messages of NM crash test

| | | **# of Ns in a NM group** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **4** | | **8** | | **16** | | **32** | |
| **# of NMs** | **2** | 2 | 1m+4 | 2 | 1m+8 | 2 | 1m+16 | 2 | 1m+32 |
| | **4** | 2 | 1m+4 | 2 | 1m+8 | 2 | 1m+16 | 2 | 1m+32 |
| | **6** | | | 2 | 1m+8 | 2 | 1m+16 | 2 | 1m+32 |
| | **8** | | | 2 | 1m+8 | 2 | 1m+16 | 2 | 1m+32 |
| | **16** | | | 2 | 1m+8 | 2 | 1m+16 | | |

### *6.1.4 CM Crash Test*

In this test, CM is stopped instantly and the operation taken by the cluster is watched. In normal operation, CM sends a Heartbeat message if it didn't not communicate with its backup NM in a certain interval (Activity time) to inform its aliveness. If CM dies, then it will not send Heartbeat messages and backup NM realizes this situation and sends a CHECKALIVE message to CM and waits an immediate reply within a CheckAlive time period. Since CM died, when this period is over, backup NM promotes itself as the new CM. When the CM thread is activated it sends a NEWCM multicast message to its group. Upon receiving that message, NMs responds it with NMOK messages. The number of messages sent during the detection and promotion of new CM is constant and 2. During the start of the new CM single multicast message is sent, but number of unicast messages depends on the number of NMs.

The time measurements are shown in Table 6.5, and amount of sent messages in Table 6.6. The first sub columns below labels are for detection and removal, the second ones are for start of the new CM. In tables CheckAlive time period is shown as CA, and amount of multicast messages are shown with an "m" at the end. Tests show that the detection and promotion operation is around CheckAlive time period and is not effected by the size and shape of the cluster. The time required for start of the new CM mainly depends on the number of NMs in the cluster. The time changes by the number of Ns are ignored since it is caused by performing the tests in a single machine.

Table 6.5 Results of CM crash test

| | | **# of Ns in a NM group** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **4** | | **8** | | **16** | | **32** | |
| **# of NMs** | **2** | CA+1 | 38 | CA+1 | 36 | CA+2 | 35 | CA+0 | 43 |
| | **4** | CA+2 | 60 | CA+1 | 62 | CA+2 | 63 | CA+0 | 64 |
| | **6** | | | CA+2 | 79 | CA+0 | 81 | CA+2 | 81 |
| | **8** | | | CA+1 | 105 | CA+4 | 103 | CA+8 | 100 |
| | **16** | | | CA+0 | 127 | CA+3 | 127 | | |

Table 6.6 Volumes of transmitted messages of CM crash test

| | | **# of Ns in a NM group** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **4** | | **8** | | **16** | | **32** | |
| **# of NMs** | **2** | 2 | 1m+2 | 2 | 1m+2 | 2 | 1m+2 | 2 | 1m+2 |
| | **4** | 2 | 1m+4 | 2 | 1m+4 | 2 | 1m+4 | 2 | 1m+4 |
| | **6** | | | 2 | 1m+6 | 2 | 1m+6 | 2 | 1m+6 |
| | **8** | | | 2 | 1m+8 | 2 | 1m+8 | 2 | 1m+8 |
| | **16** | | | 2 | 1m+16 | 2 | 1m+16 | | |

### *6.1.5 Split Operation Test*

Split operation is started by CM when it is necessary to employ a new NM and construct a new group in cluster. In this operation, CM sends a SPLIT multicast message to the top group. When an NM receive this message, it sends a NMUPDATE message to its group. Upon receiving this message Ns respond it with NOK messages. When all NOK messages are received, NM sends a NMOK message to CM to inform completion of the operation. After all NMOK messages are received, CM promotes a selected N as the NM of the new group. When the new NM thread is activated, it sends a NEWNM multicast message to its group. Upon receiving this message Ns of that group responds it with NOK messages.

The time measurements are shown in Table 6.7, and amount of sent messages in Table 6.8. The first sub columns below labels are for SPLIT and NMUPDATE sequences, the second ones are for start of the new NM. In tables volume of sent multicast messages are shown with an "m" at the end. Tests show that the split operation depends on the size of the cluster. As the number of Ns and NMs grows, the time requirement and message counts increases. The time required for start of the new NM mainly depends on the number of Ns of the new group.

Table 6.7 Results of split operation test

| | | **# of Ns in a NM group** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **4** | | **8** | | **16** | | **32** | |
| **# of NMs** | **2** | 89 | 22 | 107 | 39 | 192 | 50 | 247 | 58 |
| | **4** | 91 | 32 | 148 | 36 | 225 | 37 | 372 | 56 |
| | **6** | | | 183 | 31 | 292 | 38 | 495 | 68 |
| | **8** | | | 235 | 44 | 415 | 51 | 655 | 69 |
| | **16** | | | 376 | 38 | 740 | 117 | | |

Table 6.8 Volumes of transmitted messages of split operation test

| | | **# of Ns in a NM group** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **4** | | **8** | | **16** | | **32** | |
| **# of NMs** | **2** | 3m+15 | 1m+6 | 3m+27 | 1m+13 | 3m+51 | 1m+26 | 3m+99 | 1m+53 |
| | **4** | 5m+16 | 1m+7 | 5m+30 | 1m+13 | 5m+56 | 1m+27 | 5m+110 | 1m+54 |
| | **6** | | | 7m+37 | 1m+15 | 7m+71 | 1m+30 | 7m+138 | 1m+61 |
| | **8** | | | 9m+46 | 1m+16 | 9m+87 | 1m+33 | 9m+169 | 1m+66 |
| | **16** | | | 17m+81 | 1m+20 | 17m+154 | 1m+41 | | |

### *6.1.6 Summary of Test Results*

The tests show the following results:

- N join and failure operations are simple and performance is not effected by the size of the cluster, hence they are not a scalability issue.

- NM failover operation is handled by effecting only the group where problem occurs and isolated from other parts of the cluster. the operation does not cause scalability problems.

- CM failover operation is handled in the management group so Ns are not effected. Operation does not have scalability problems.

- Split operation and constructing a new group is a process that effects the whole cluster. Tests show that amount of message transmits and operation times increase as the cluster grows. Considering the message sizes and volume increase ratio, split operation is not a dramatic scalability issue on a fast and reliable network environment. Besides, the time period of operation is effected mainly by the simulation environment since tests are performed on a single machine environment.

## 6.2 Tests For Load Balancing Model

The implementation of the load balancing model was tested on a simulation environment. The tests performed aim as a proof of efficiency and scalability of the model, observation of its functionality and performance, while discovering possible improvements.

The program code of CIM protocol was written with C language for Linux environment and compiled on a kernel version 2.6.3 Mandrake Linux 10.0 (x86) operating system. Since there are not sufficient resources yet for construction of a real cluster environment (tens or even hundreds of workstations are needed), a single machine was used in simulation. For simulation, multiple Ns, NMs and a CM module thread virtually representing different workstations, run on this machine in a single CIM process. The machine was a PC with Intel Centrino Duo 1.8 Ghz processor and 1GB memory.

In simulation two load indices was used with equal weights of 0.5: cpu utilization and memory utilization. Then the calculated load value of a N is:

$$L = l_{cpu} \text{ x } 0.5 + l_{memory} \text{ x } 0.5$$

As load index thresholds, 0.8 was set for both indices. That means when a load index value exceeds 80%, that load index effects directly to calculated load value regardless of its weight to prevent resource overflow.

In this environment, each N was assigned a set of resources at startup randomly from a resource table. The resource table is shown in Table 6.9. The resource table was used to simulate the heterogeneity of Ns.

Table 6.9 Heterogeneous resources used in tests

| **Resource ID** | **Cpu** | **Memory** |
|---|---|---|
| 0 | 1 | 128 |
| 1 | 1 | 256 |
| 2 | 1.5 | 256 |
| 3 | 1.5 | 512 |
| 4 | 2 | 256 |
| 5 | 2 | 512 |

Since we assumed that task arrival times, rates and durations were not known at runtime, each N was submitted randomly selected tasks from a task table, at random times dynamically. The execution duration for each task was also a random time from 20 seconds to 100 seconds. Each task has a specific resource usage. The cpu usage of a task represents its  average cpu consumption on a base computer which has resources as resource ID 0, whereas memory usage represents absolute memory consumption while the task is running. Task table is shown in Table 6.10.

Table 6.10 List of task profiles used in simulations

| **Task ID** | **Cpu consumption** | **Memory consumption** |
|---|---|---|
| 0 | 15 | 8 |
| 1 | 13 | 12 |
| 2 | 15 | 11 |
| 3 | 12 | 18 |
| 4 | 17 | 17 |
| 5 | 8 | 9 |
| 6 | 10 | 20 |
| 7 | 8 | 16 |
| 8 | 15 | 7 |
| 9 | 6 | 13 |

In simulations, non-preemptive task transfers were considered. So, tasks that are newly submitted and  not started were involved in task transfers.

In tests, the load balancing model run on clusters with 2,4 and 6 NMs having 4, 8, 16 Ns. In the tests, some load balancing metrics and runtime load state values was measured over a time period. The measured values were:

- Average load values of Ns, NMs and the cluster
- $L_A$, $T_R$ and $T_S$ values calculated by NMs and CM
- Ratio of locally executed tasks (tasks that run on submitted Ns)
- Ratio of locally transferred tasks (tasks that run on different Ns in a NM group, other than where they are submitted)
- Ratio of globally transferred tasks (tasks that were sent to other Ns that belong to other NM groups)

- Ratio of globally imported tasks (tasks that were received from other Ns that belong to other NM groups)

In Figure 4, the amount of load submitted to the cluster systems during tests are shown (cluster names represent their sizes, e.g. 2x8 cluster has 2 NMs each having 8 Ns). The volume of tasks submitted were proportional to the number of Ns clusters had.

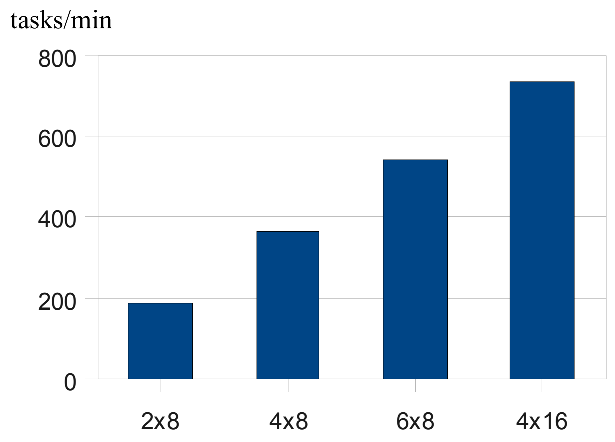


Figure 6.1 Tasks submission rates for different cluster sizes.

### *6.2.1 Test 1: 2x8 Cluster*

In this test a 2 NMs with 8 Ns cluster was constructed on which local and global load balancing operations measured. The test results in Figures 6.2-6.3 show that local load balancing scheme was able to keep load levels of highly utilized Ns around $T_S$ levels by transferring their submitted tasks to the Ns that have low load and prevented overloading. NMs adjusted the $T_R$ and $T_S$ threshold values around load averages to specify load senders and receivers. This also prevented moderately loaded (close to the average) Ns to be involved in load transfers that cause useless task transfers. So, task transfers were performed from highly loaded Ns through lightly loaded Ns. As the graphs in Figure 6.4-6.5 show, Ns mainly run local tasks and average task transfer rates were kept low, avoiding transfer overheads.

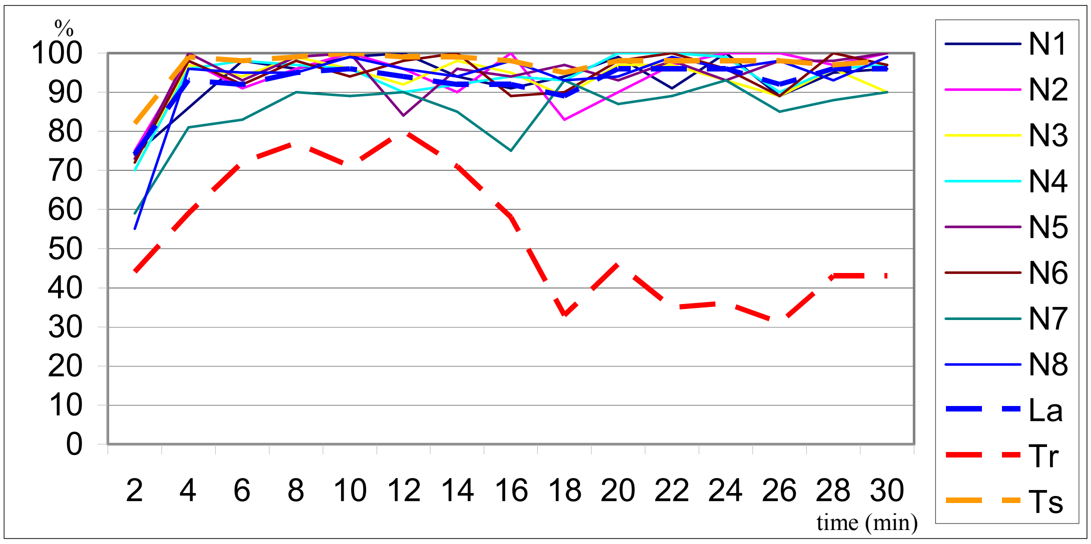


Figure 6.2 Local load values of NM1 group.

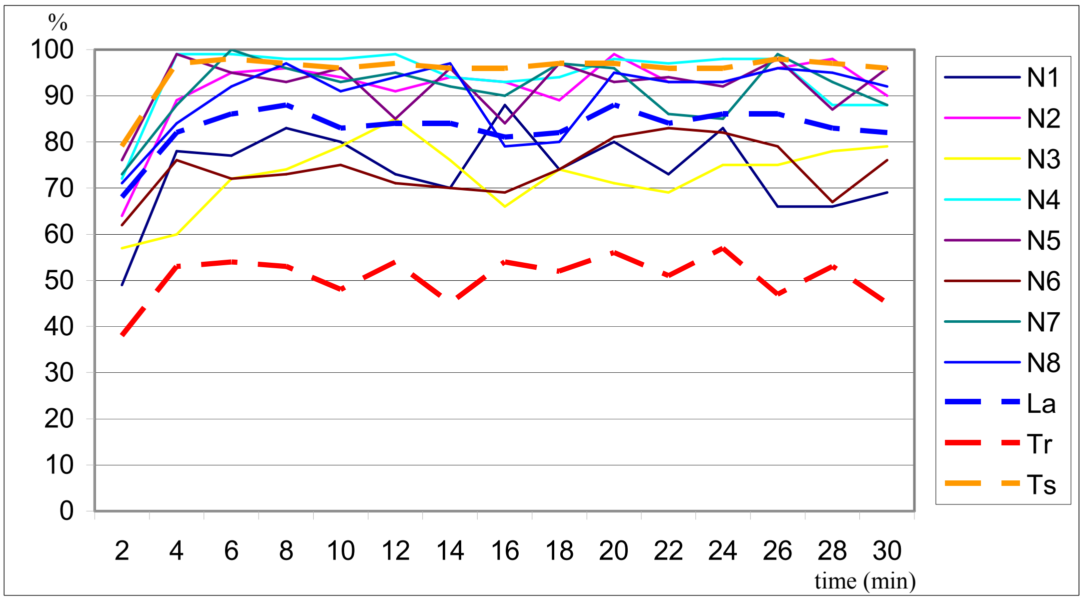


Figure 6.3 Local load values of NM2 group.

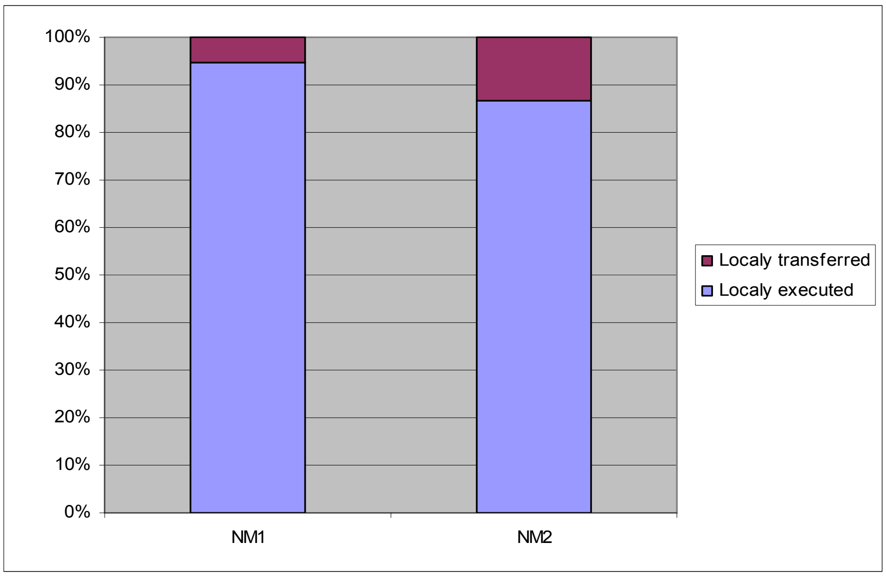


Figure 6.4 Ratio of locally executed and transferred tasks.

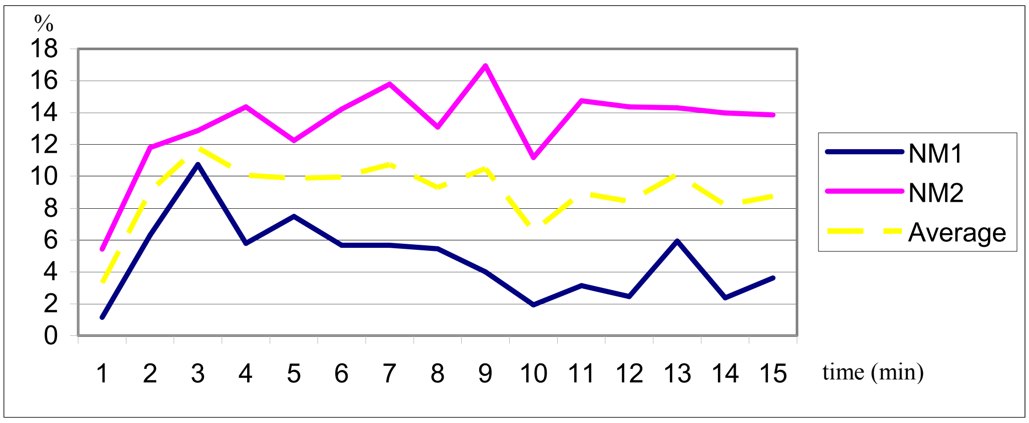


Figure 6.5 Rates of locally transferred tasks by time period.

As seen in Figure 6.6 both NM groups are highly loaded, so global load sharing module had less chance to find receiver NM group. Therefore, as shown in Figure 6.7-6.9 although global load transfer rates are very low, few load transfers were performed from NM1 to NM2 which had less load then the other.

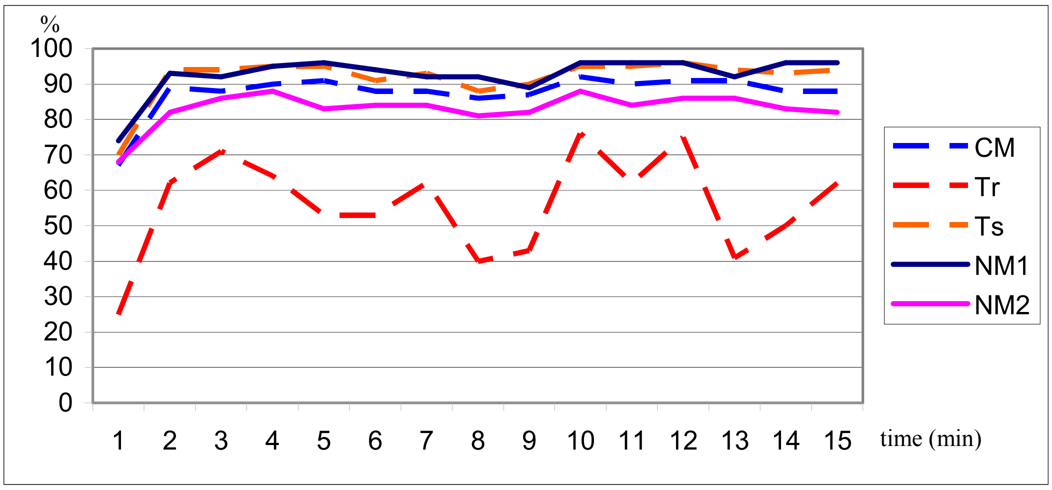


Figure 6.6 Global load values.

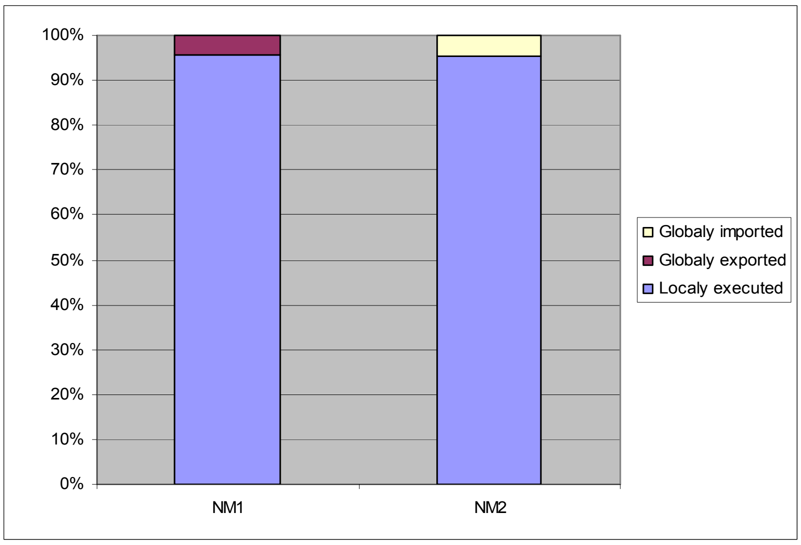


Figure 6.7 Ratio of locally executed and globally transferred tasks.

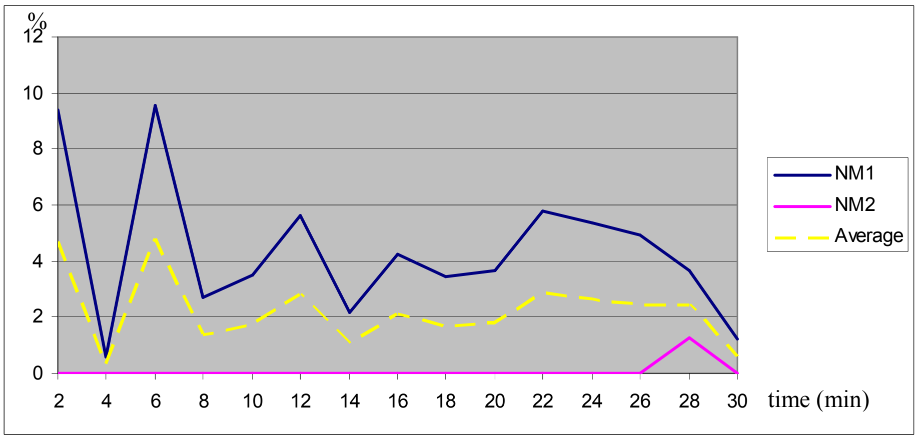


Figure 6.8 Rates of globally exported tasks by time period.

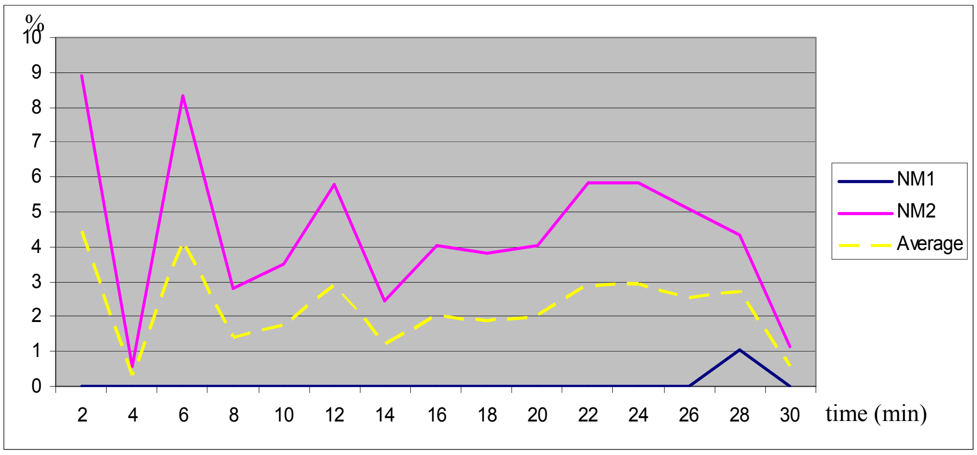


Figure 6.9 Rates of globally imported tasks by time period.

### *6.2.3 Test 2: 4x8 Cluster*

In this test a 4 NMs with 8 Ns cluster was constructed and load balancing operations are measured. The test results were shown in Figures 6.10-6.15. The results show similar results with previous test. The local load balancing model was able to keep load levels of highly utilized Ns around $T_S$ levels. $T_R$ and $T_S$ threshold values were adjusted according to the load values. This prevented moderately loaded Ns to be involved in load transfers that would cause useless task transfers. Again, average task transfer rates were kept low.

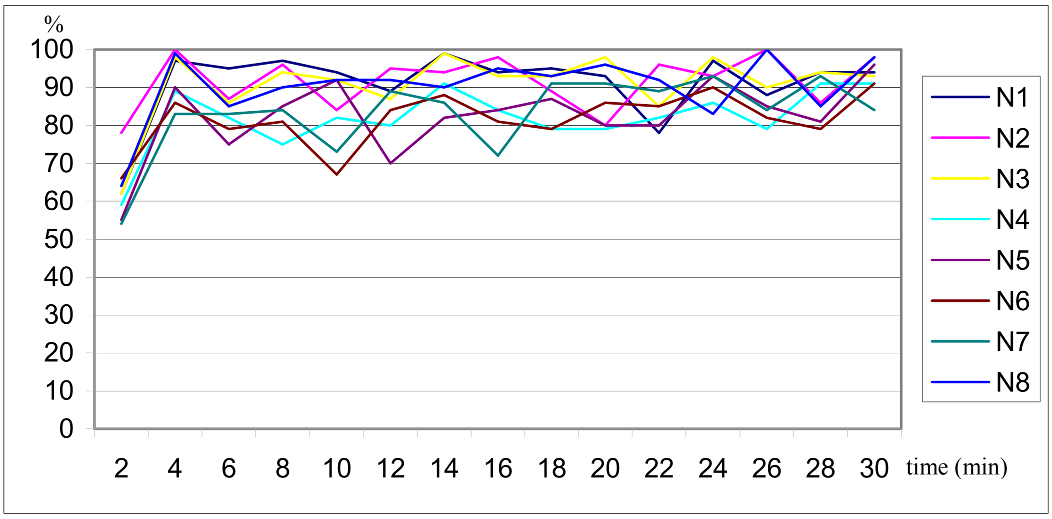


Figure 6.10 Local load values of NM1 group.

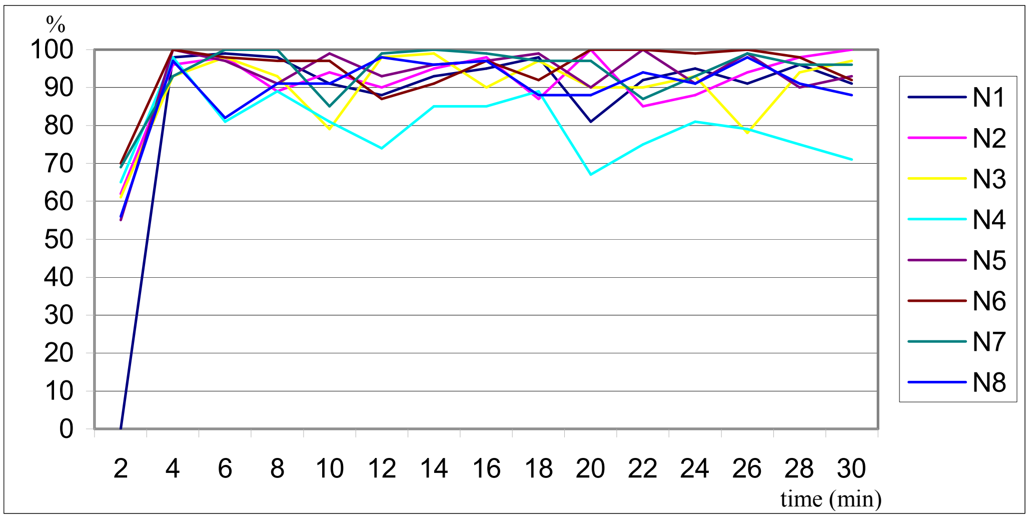


Figure 6.11 Local load values of NM2 group.

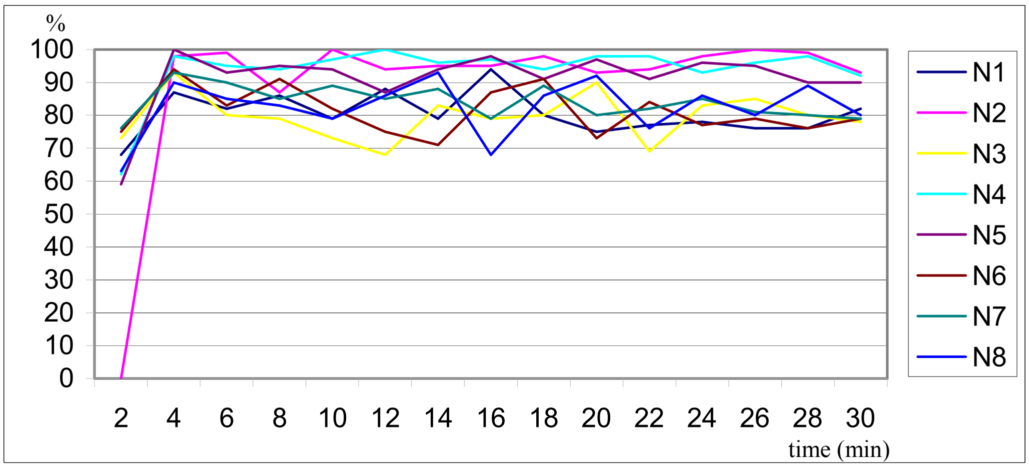


Figure 6.12 Local load values of NM3 group.

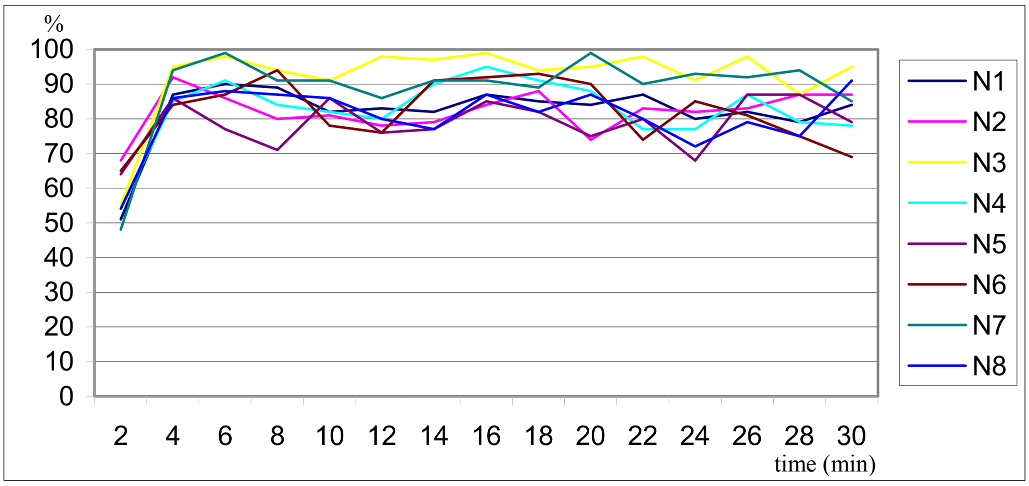


Figure 6.13 Local load values of NM4 group.

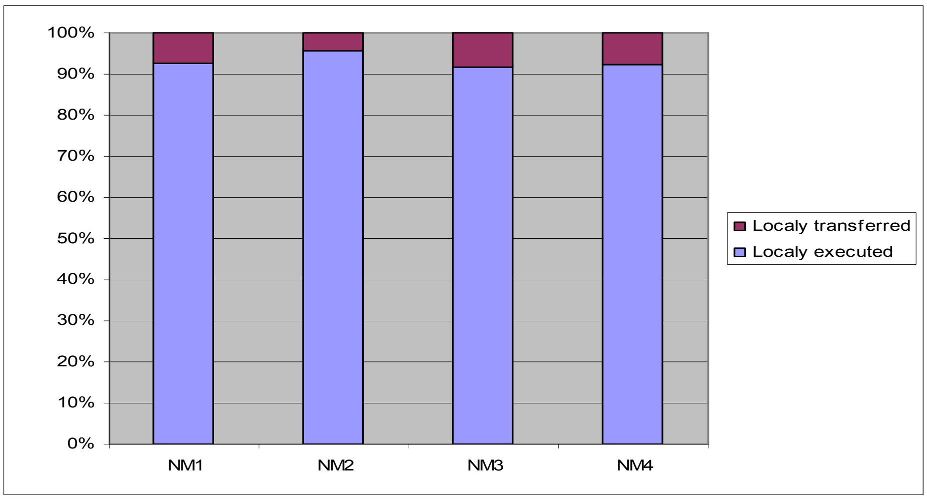


Figure 6.14 Ratio of locally executed and transferred tasks.

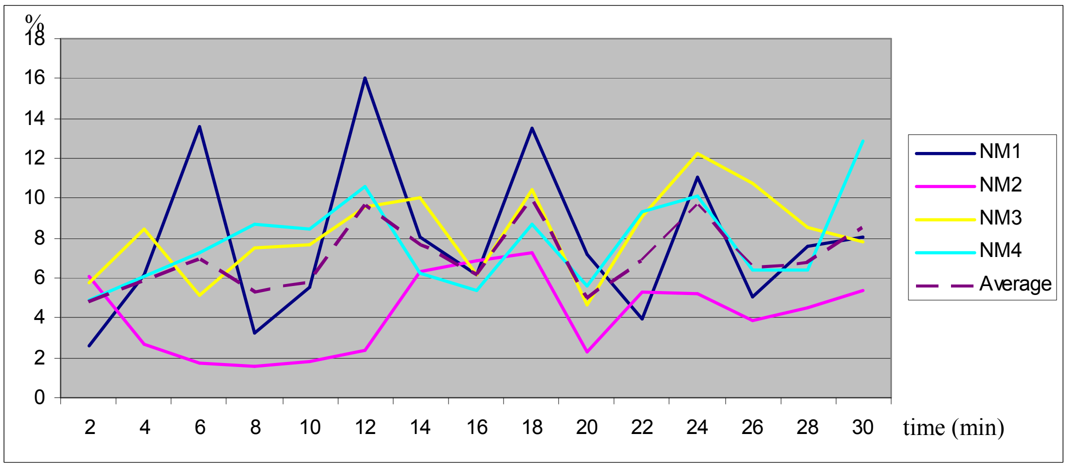


Figure 6.15 Rates of locally transferred tasks by time period.

Figure 6.16-6.19 show the global load balancing operation results. Group load levels are around average cluster load. Global task transfers are mainly performed through highly load groups like NM2 and NM1 to the receiver groups NM3 and NM4. The task transfer rates were below %5.

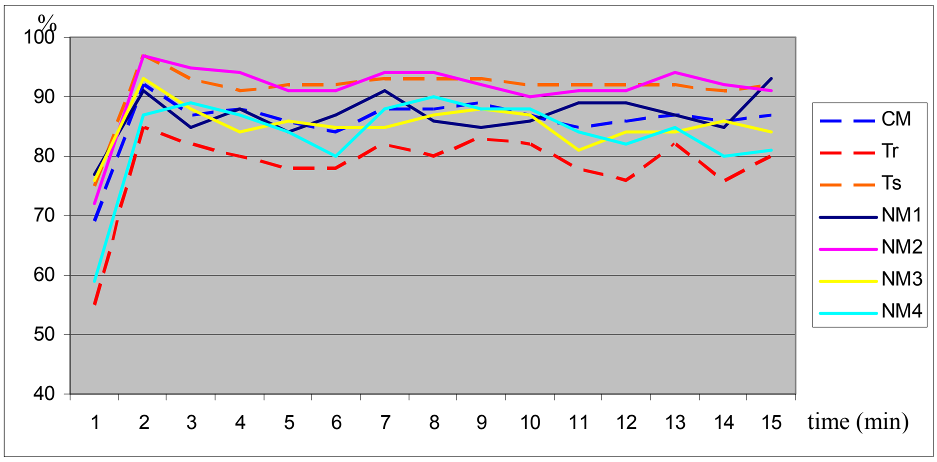


Figure 6.16 Global load values.

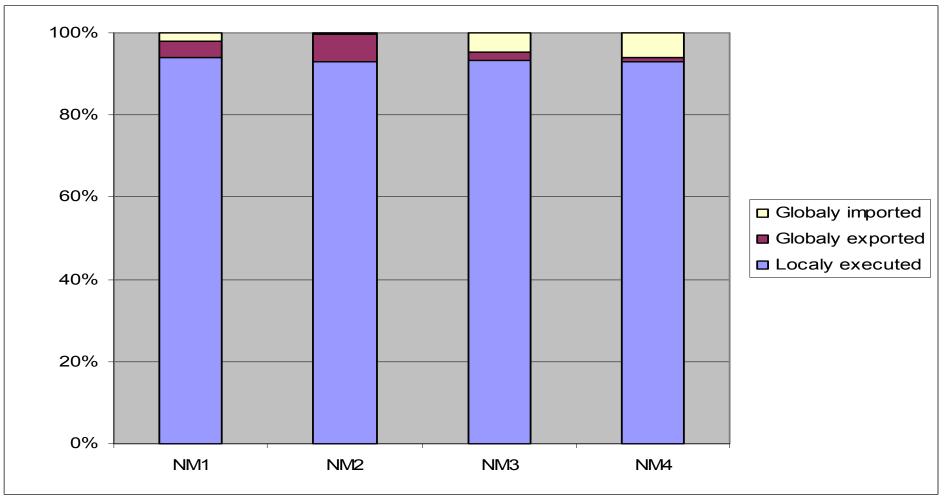


Figure 6.17 Ratio of locally executed and globally transferred tasks.

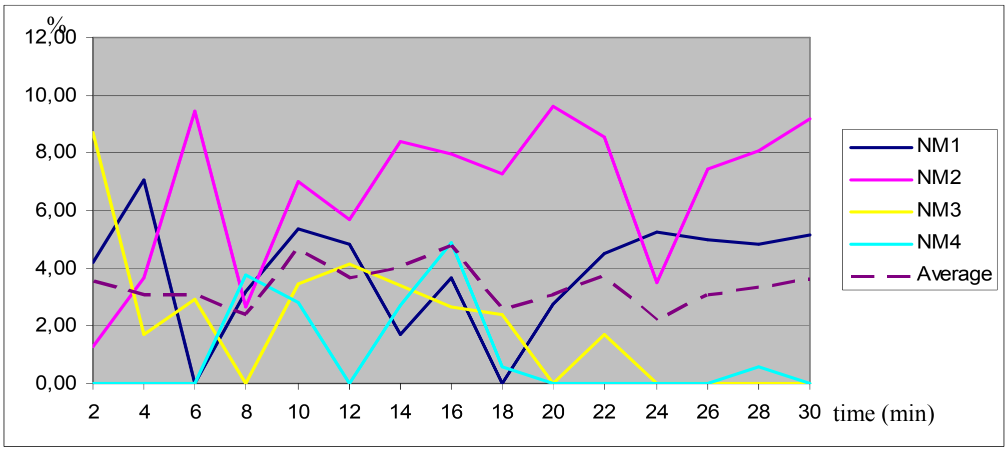


Figure 6.18 Rates of globally exported tasks by time period.

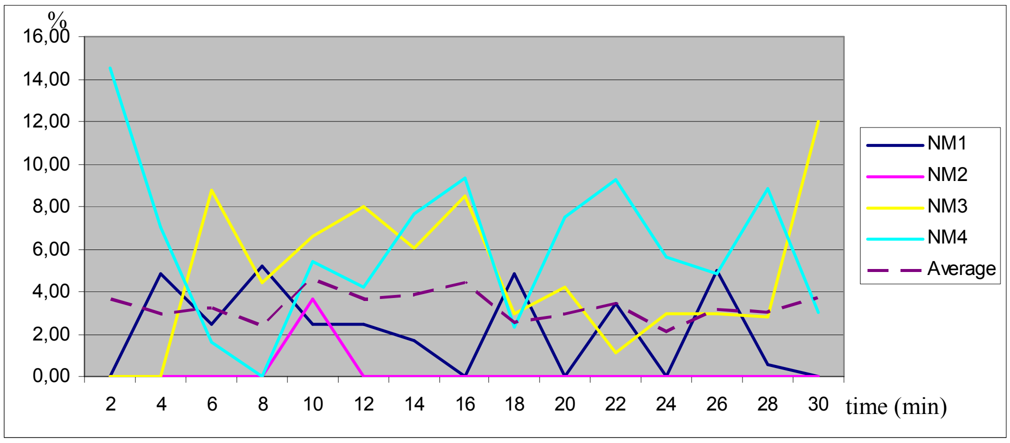


Figure 6.19 Rates of globally imported tasks by time period.

### *6.2.4 Test 3: 6x8 Cluster*

In this test a 6 NMs with 8 Ns cluster was constructed and local load balancing operations are measured. The test results were shown in Figures 6.20-6.31. The comments of the previous tests can also be stated for these results. While local load sharing policy distributes the load inside the groups, global load sharing policy transfers more load to NM3 group which has less load than others. Again, the global task transfer rates were around %5.

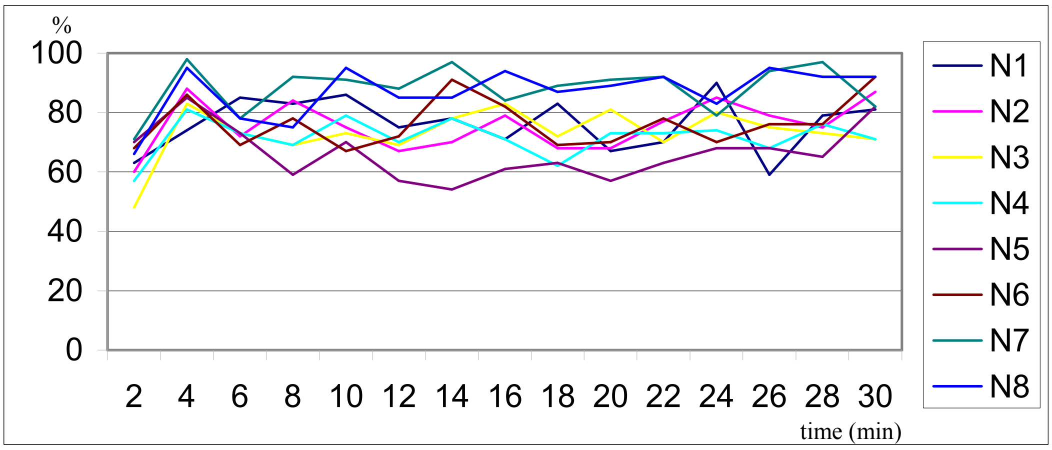


Figure 6.20 Local load values of NM1 group.

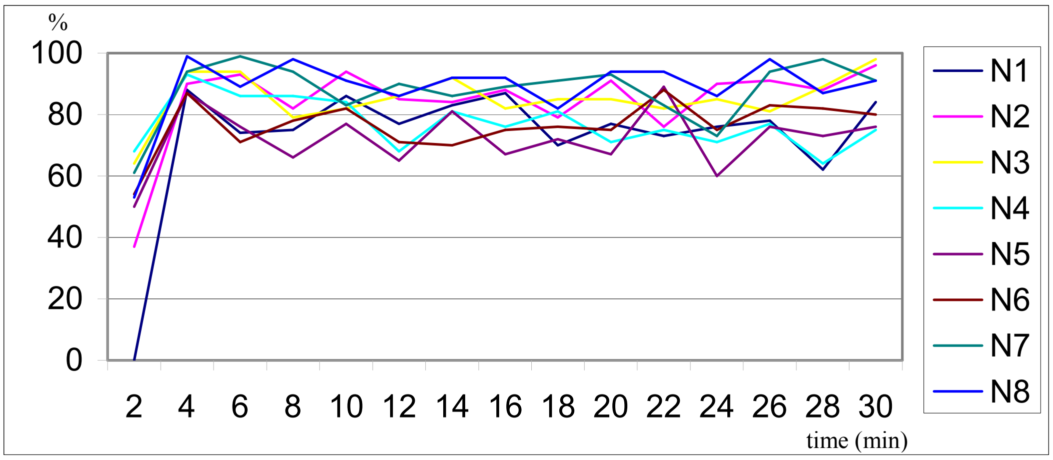


Figure 6.21 Local load values of NM2 group.

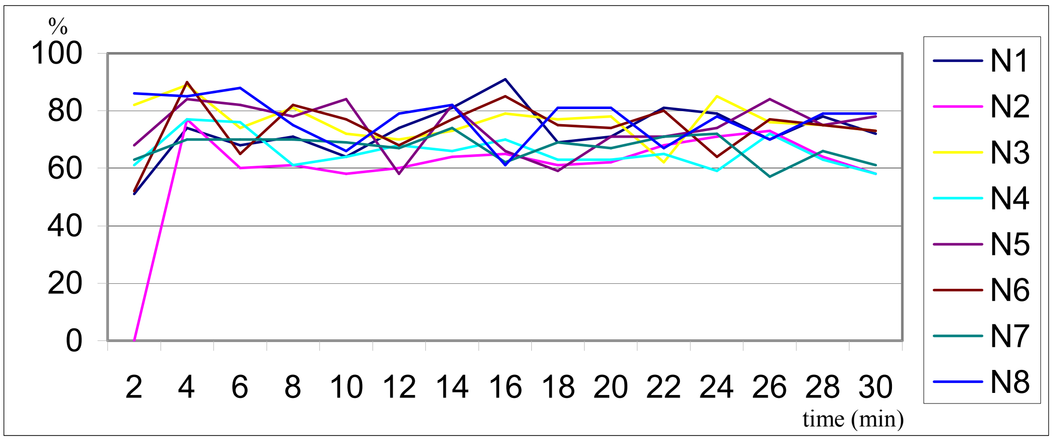


Figure 6.22 Local load values of NM3 group.

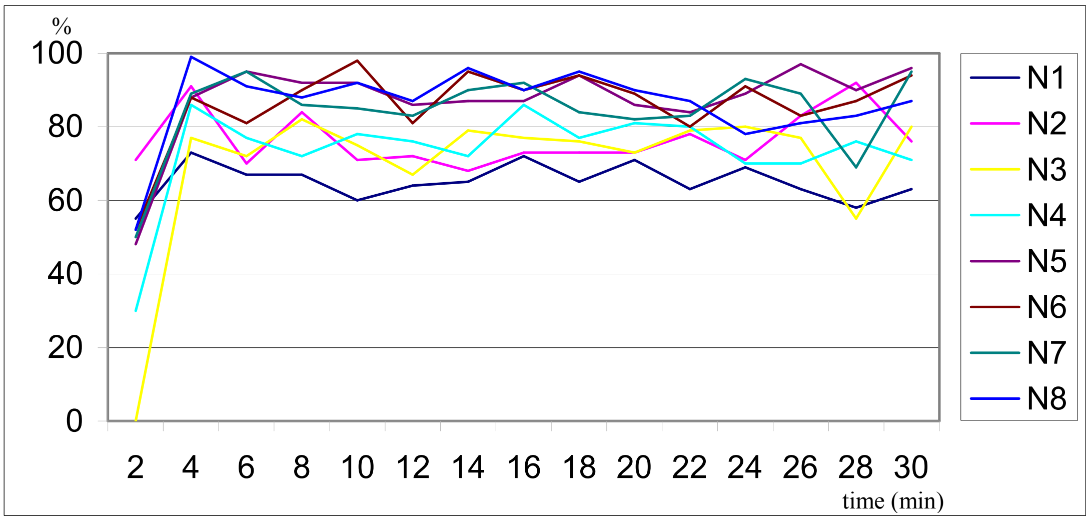


Figure 6.23 Local load values of NM4 group.

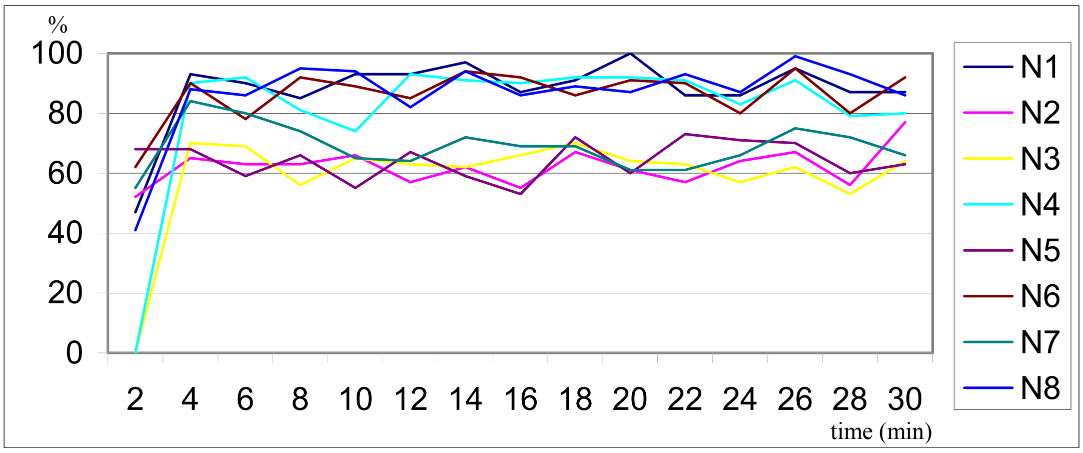


Figure 6.24 Local load values of NM5 group.

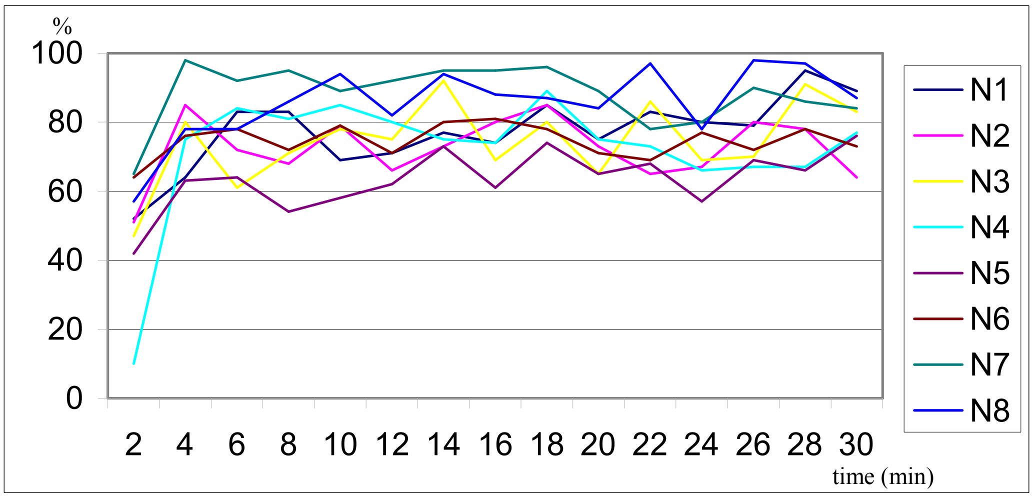


Figure 6.25 Local load values of NM6 group.

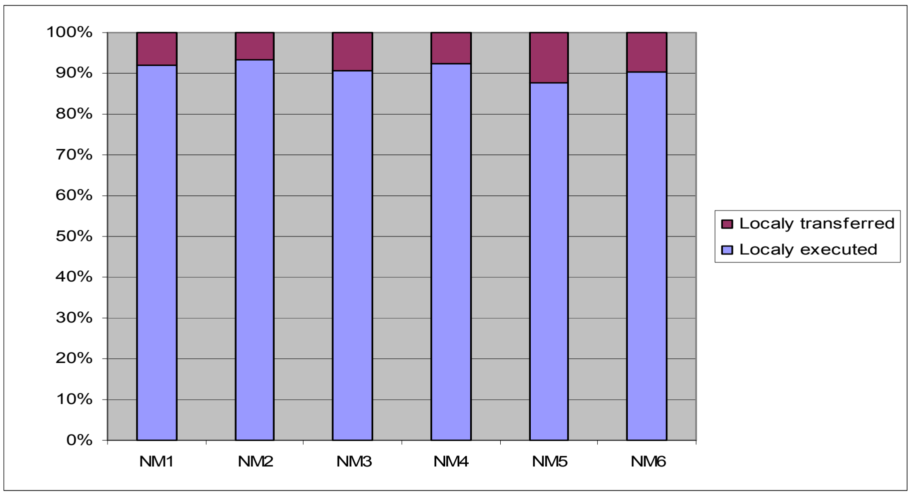


Figure 6.26 Ratio of locally executed and transferred tasks.

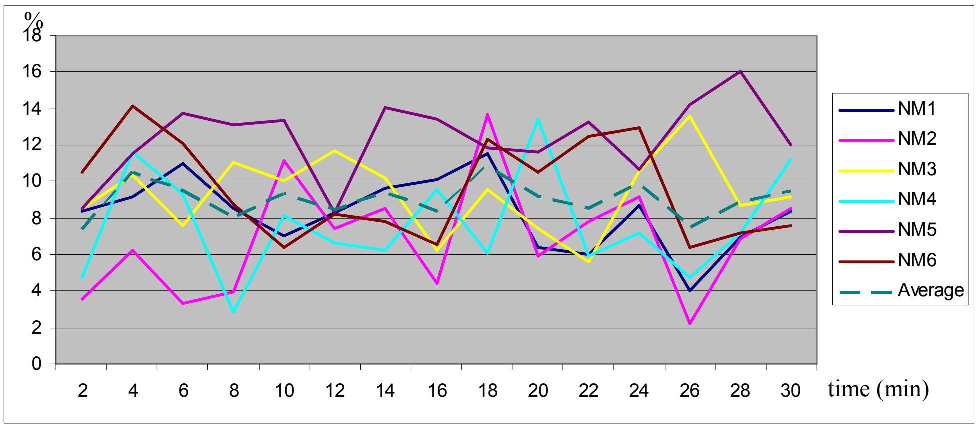


Figure 6.27 Ratio of locally executed and transferred tasks.

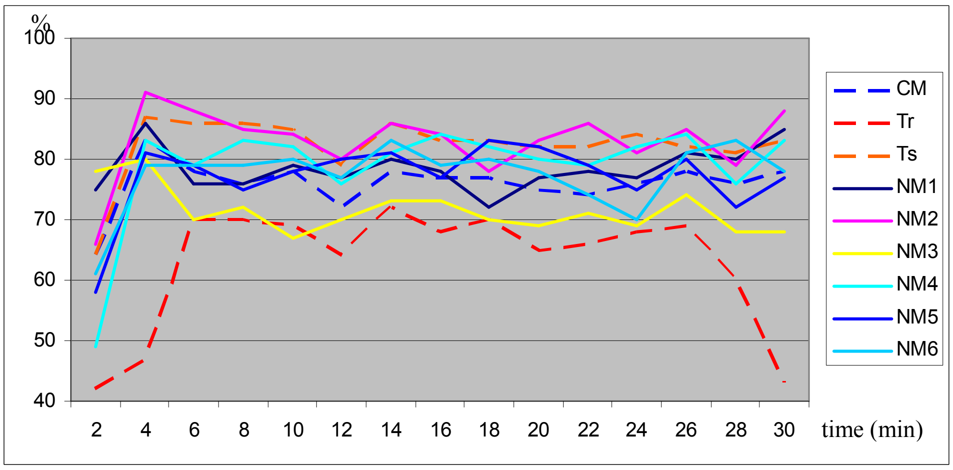


Figure 6.28 Global load values.

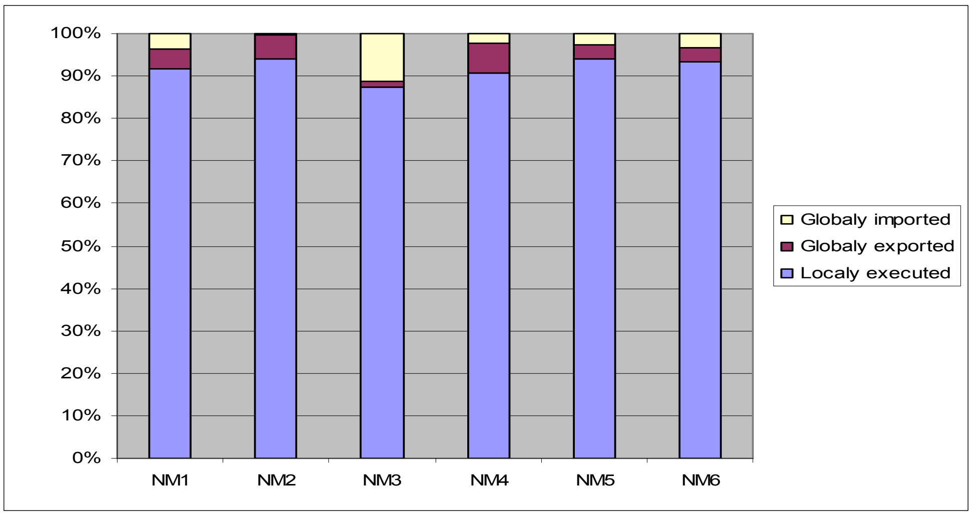


Figure 6.29 Ratio of locally executed and globally transferred tasks.

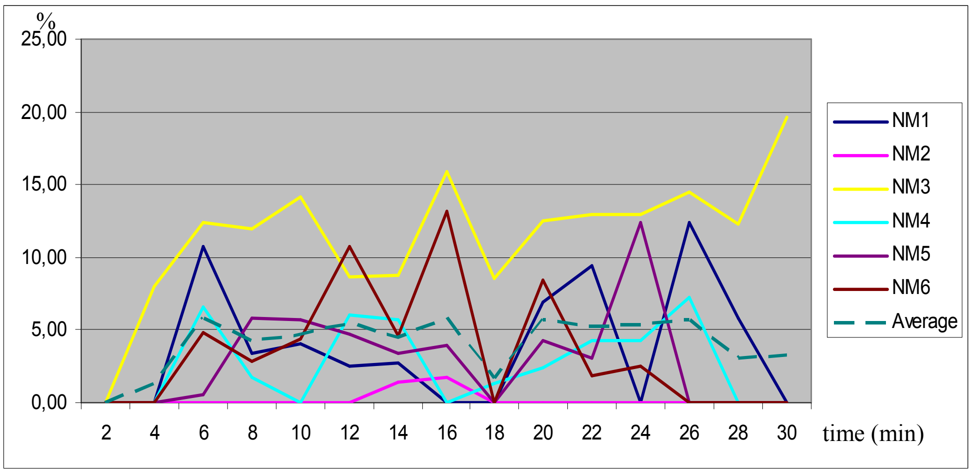


Figure 6.30 Rates of globally exported tasks by time period.

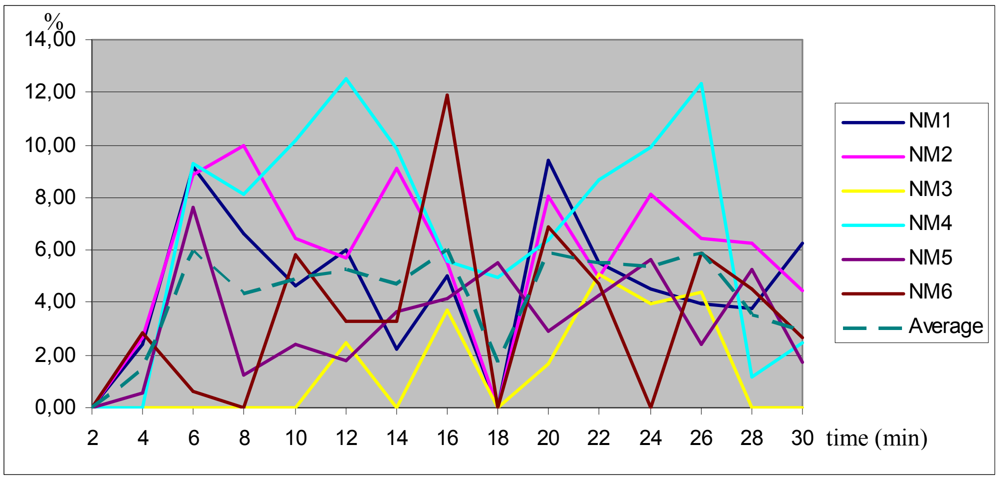


Figure 6.31 Rates of globally imported tasks by time period.

### *6.2.5 Test 4: 4x16 Cluster*

In this test a 4 NMs with 16 Ns cluster was constructed and local load balancing operations are measured. The test results for local load balancing are shown in Figures 32-37. The results show that the local load sharing policy tries to bring load values closer to the average load, by limiting sender and receiver N load values around threshold values. This also reduces task transfer rates, since the model does not try to equalize load levels. Task transfer rates were almost same low levels as in the other tests, which proves the scalability of the system.

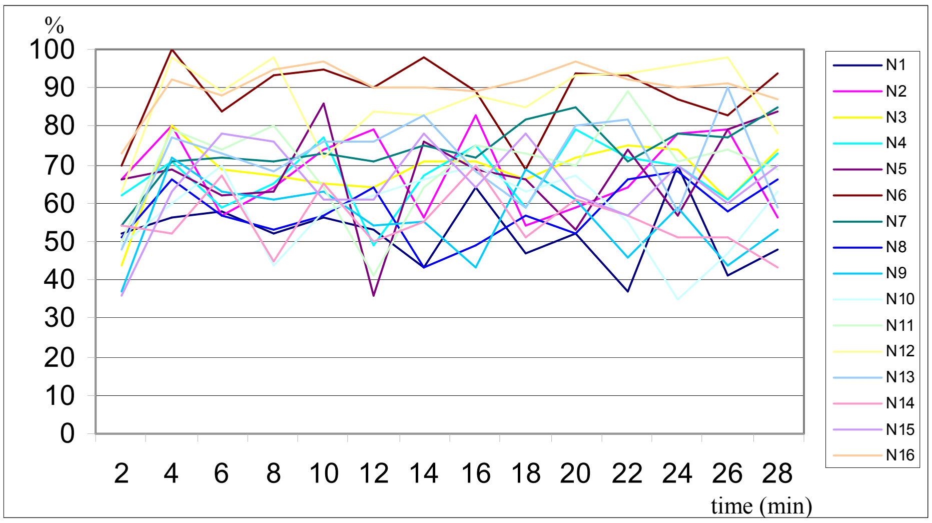


Figure 6.32 Local load values of NM1.

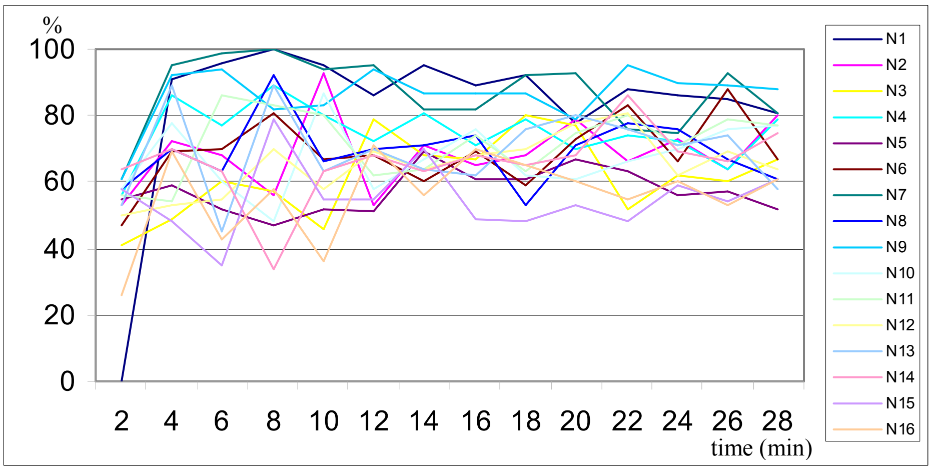


Figure 6.33 Local load values of NM2.

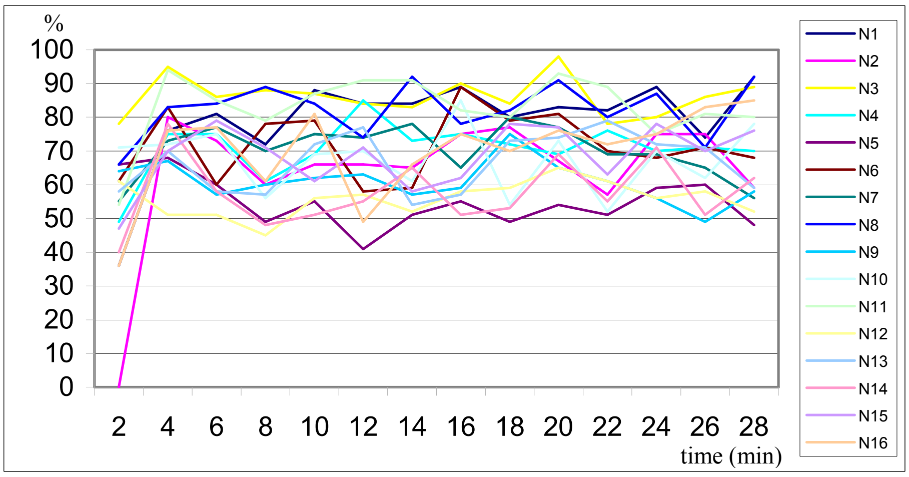


Figure 6.34 Local load values of NM3.

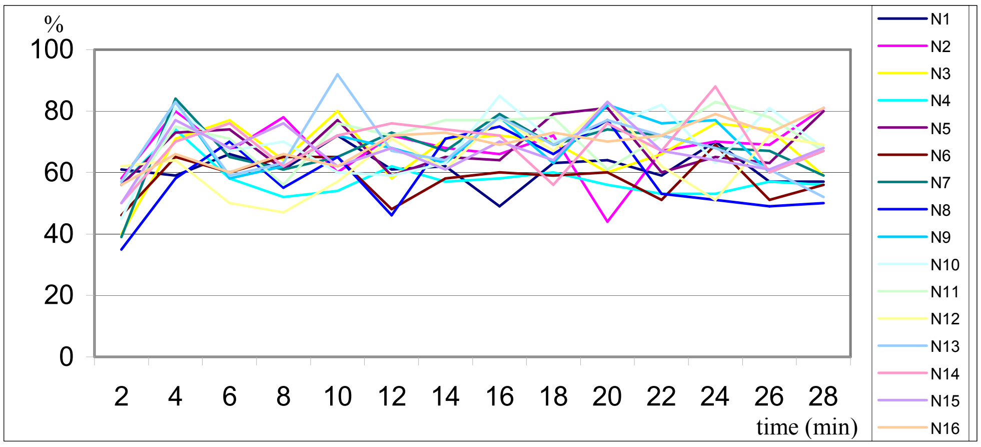


Figure 6.35 Local load values of NM4.

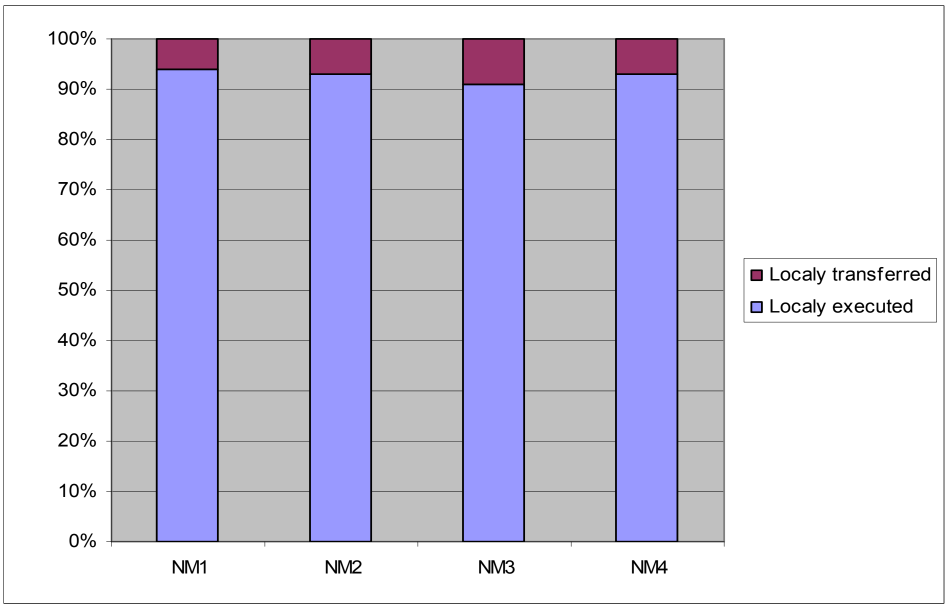


Figure 6.36 Ratio of locally executed and transferred tasks.

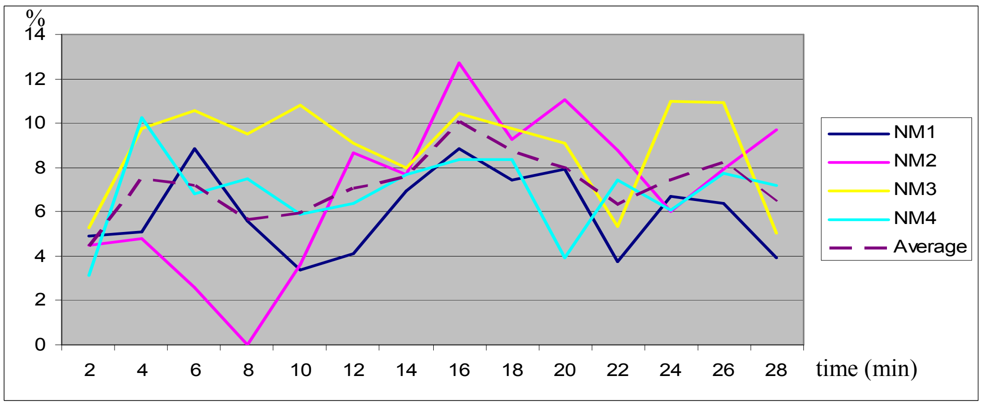


Figure 6.37 Rates of locally transferred tasks by time period.

Global load sharing policy brought the group load levels closer to the average around Ts and Tr values. Figures 38-41 shows the results of global policy. NM4 group which has least load received more tasks than the others. The top senders were NM2 and NM3. Similar to the former tests the average task transfer rate was around 3%.

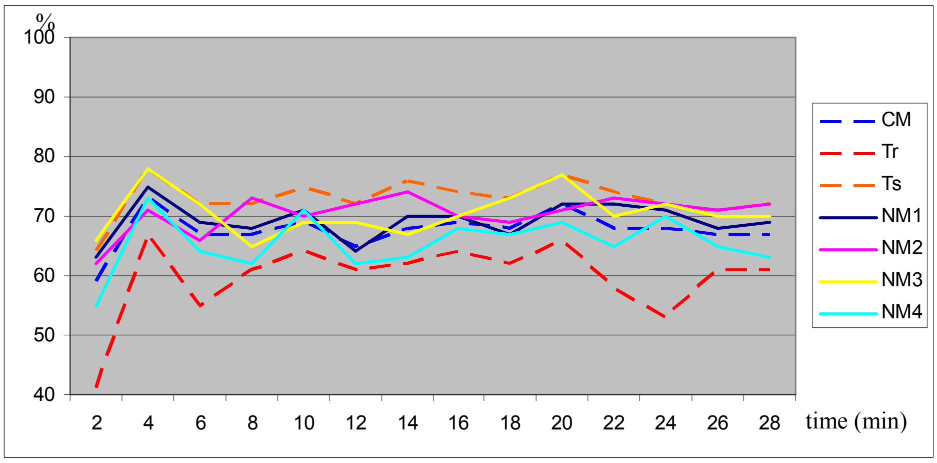


Figure 6.38 Global load values.

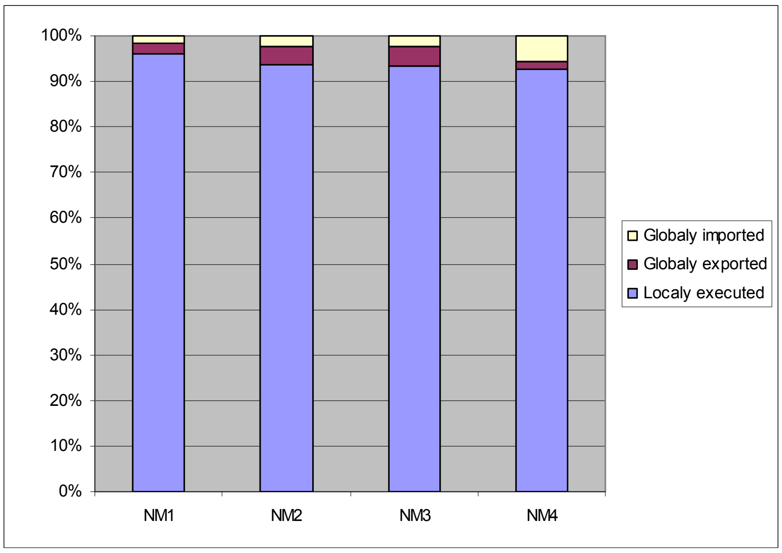


Figure 6.39 Ratio of locally executed and globally transferred tasks.

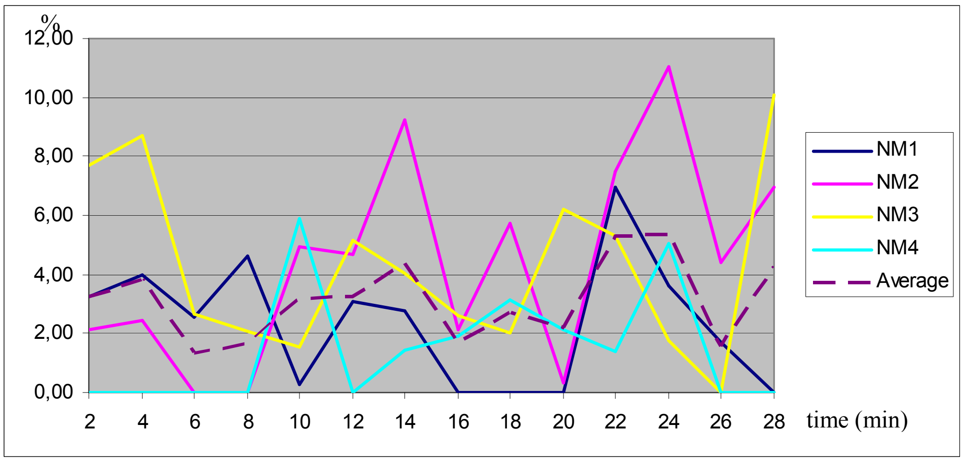


Figure 6.40 Rates of globally exported tasks by time period.

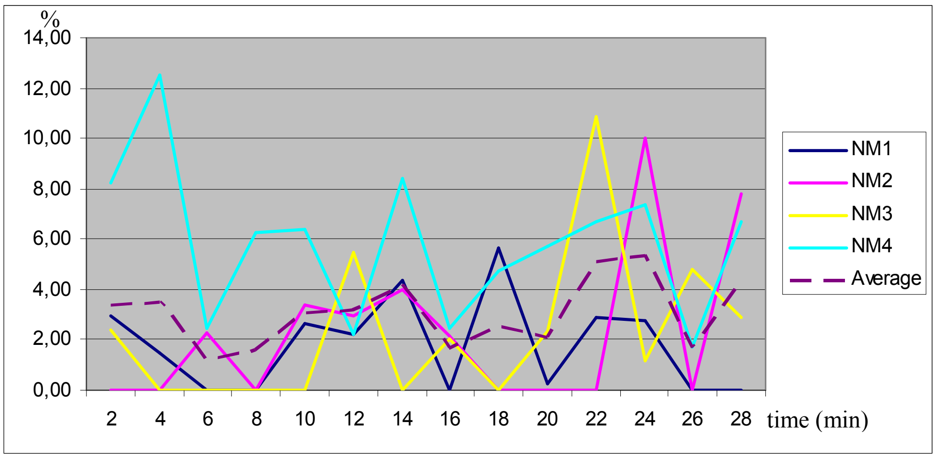


Figure 6.41 Rates of globally imported tasks by time period.

### *6.2.6 Test 5: 4x16 Cluster without Global Load Sharing*

To see the effect of global load sharing, the 4x16 cluster test were performed again with global policy disabled. Since the local policy was active, load levels inside groups were adjusted around threshold levels as shown in Figure 6.42-6.45.

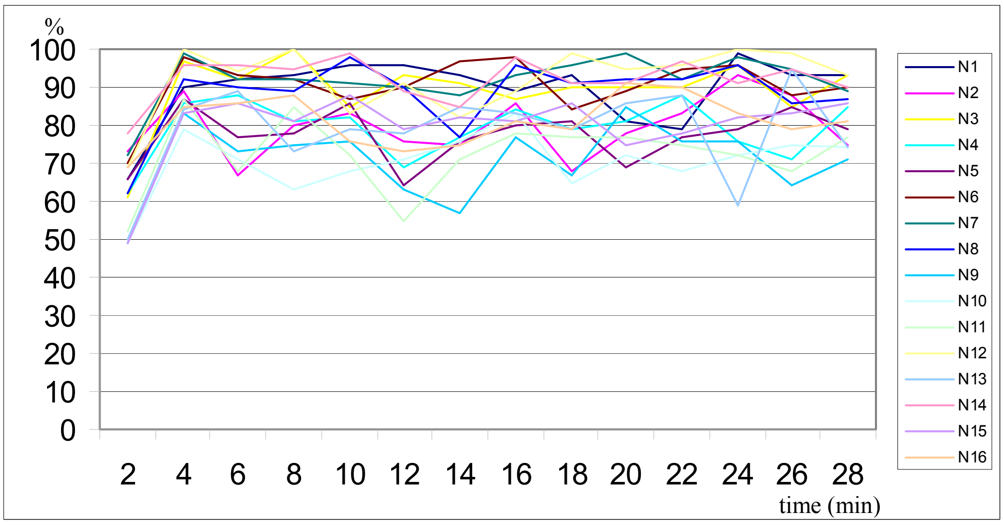


Figure 6.42 Local load values of NM1 group.

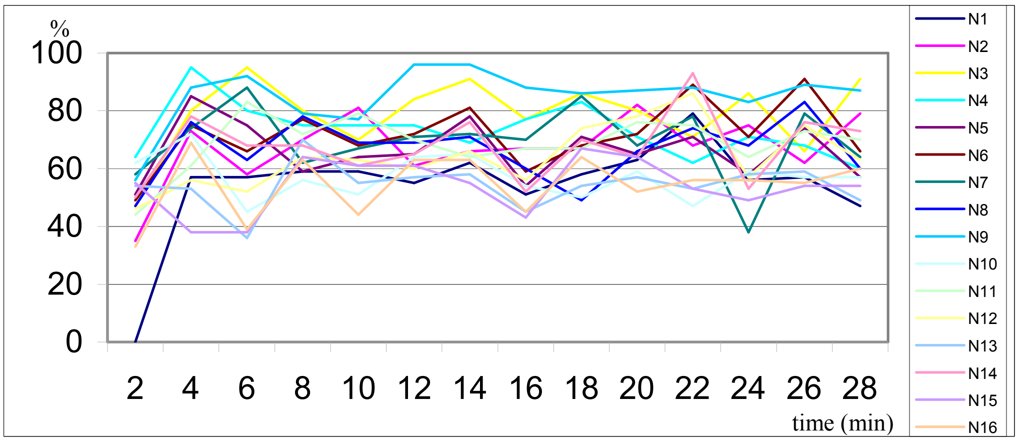


Figure 6.43 Local load values of NM2 group.

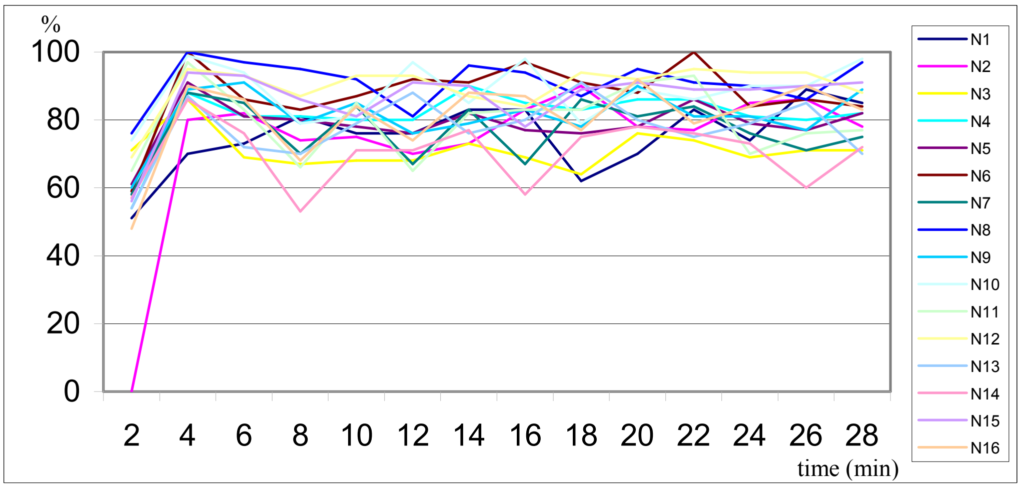


Figure 6.44 Local load values of NM3 group.

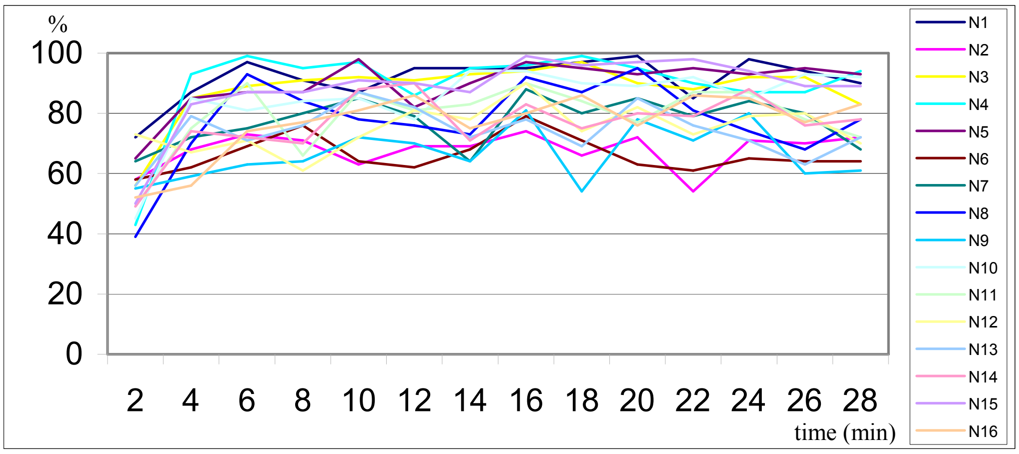


Figure 6.45 Local load values of NM4 group.

Figure 6.46 shows the group load values. Comparing this graph with Figure 6.38, the importance of global load sharing policy can be proven. With global policy, average cluster load were around %70 whereas without it the load level arises to %80, stating that the global load sharing policy distributes the loads of higher groups through groups that have less load levels and utilizes the efficient use of the cluster resources.

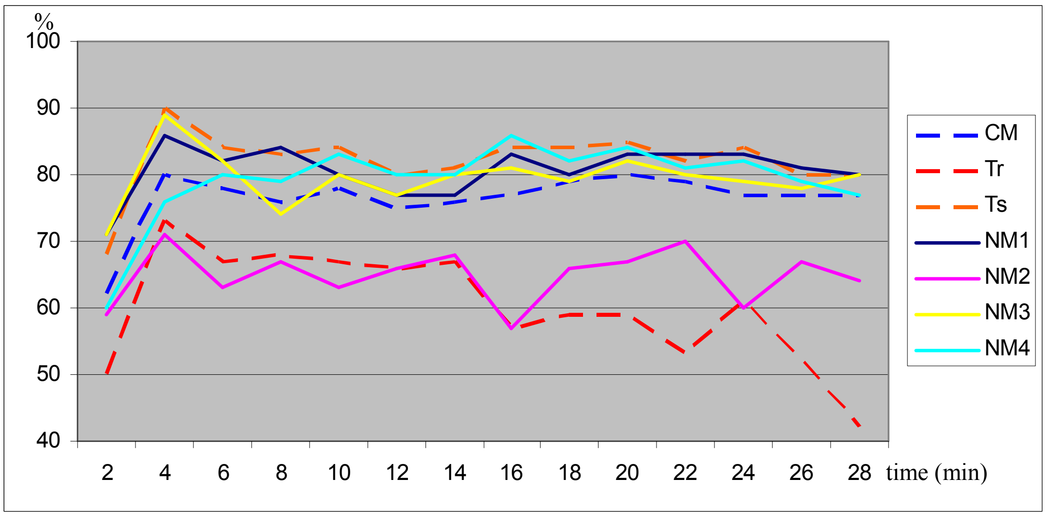


Figure 6.46 Global load values.

### *6.2.7 Test 6: 4x16 Cluster without Load Sharing*

To show the effect of load balancing model, the 4x16 test performed again with the same set of task pattern and the same resources but without the local and global load sharing policies. Figure 6.47-6.50 show the load levels inside groups. Some Ns were overloaded while there are lightly loaded Ns with the absence of local load sharing policy.

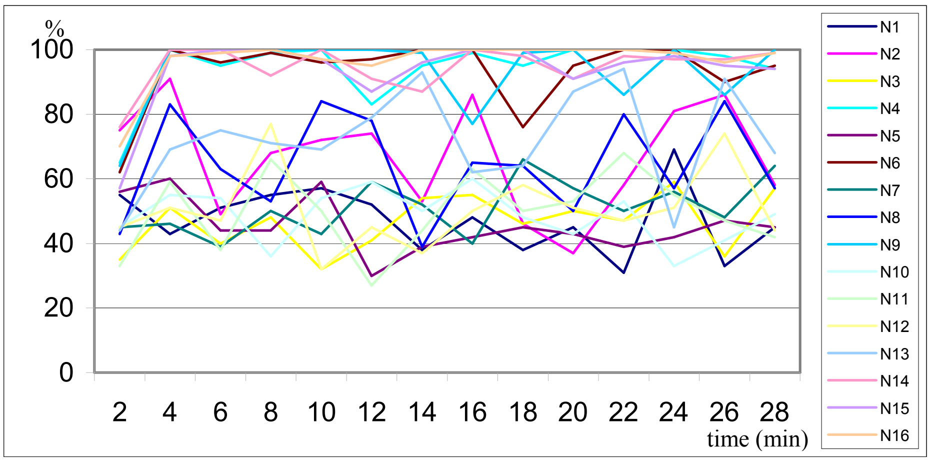


Figure 6.47 Local load values of NM1 group.

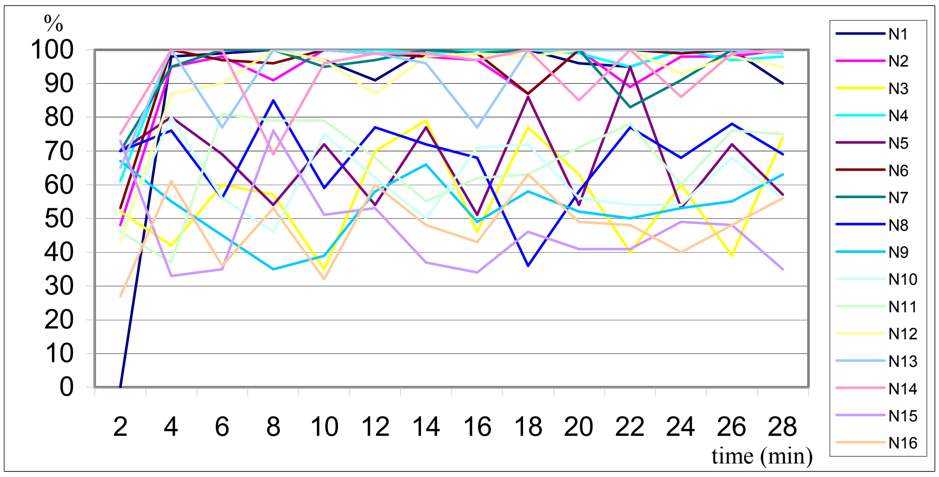


Figure 6.48 Local load values of NM2 group.

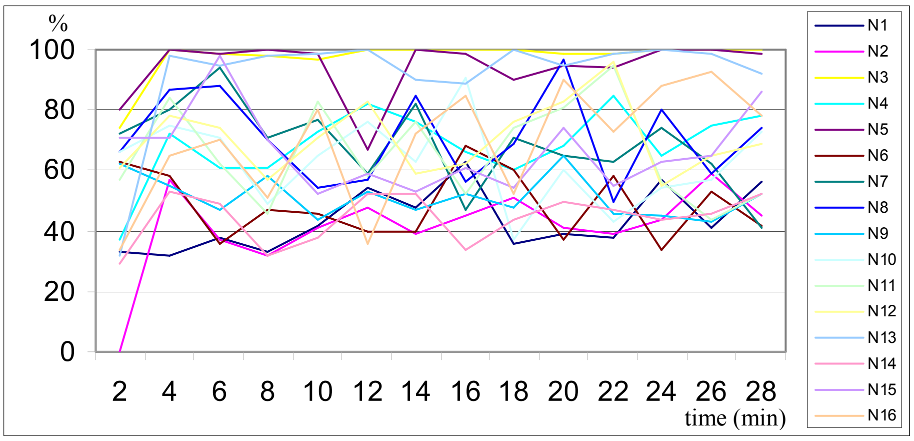


Figure 6.49 Local load values of NM3 group.

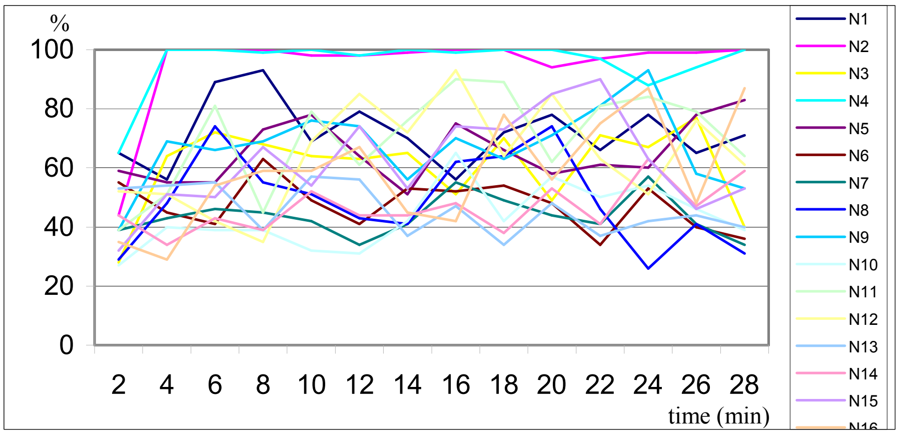


Figure 6.50 Local load values of NM4 group.

Figure 6.51 shows the unbalanced state of the cluster. As a result, the cluster resources were not efficiently utilized without the load balancing model.

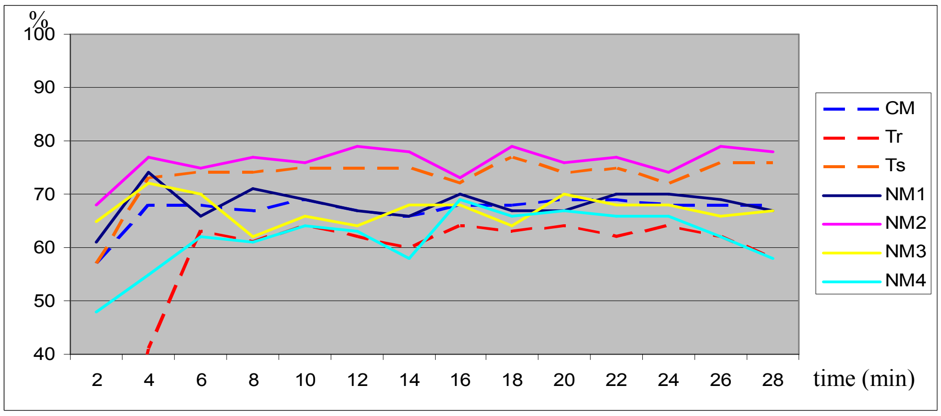


Figure 6.51 Global load values.

### *6.2.8 Evaluation of Results*

Figure 6.52 and 6.53 show average task transfer rates per minute and percentage of transferred tasks over the totals measured for both local and global load sharing policies. As naturally expected, task transfer rates increased with the growth of the cluster and with the increasing load. Global load transfers also increased by the addition of new NMs to the cluster. Note that local transfer rates are always higher than global transfer rates since independent local load sharing processes run simultaneously within the groups of Ns, while global load sharing runs on the upper level among NMs. Due to the adaptive threshold mechanism, we do not expect such differentiations as transfer rates on their percentages. In fact, test results confirms the expectations. While local transfer ratios fell below %10, global transfer percentages did not exceed %5, as seen in the Figure 6.53.

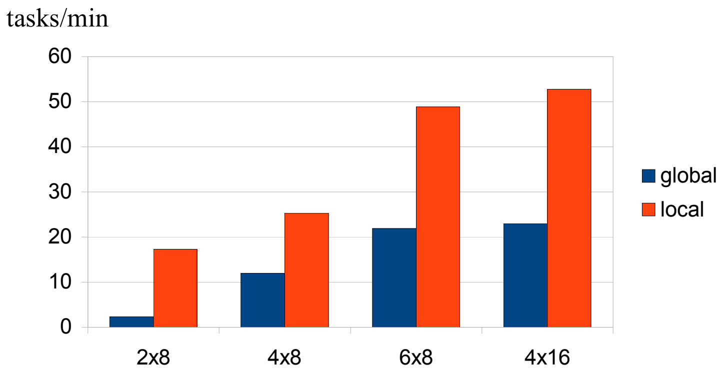


Figure 6.52 Task transfer rates.

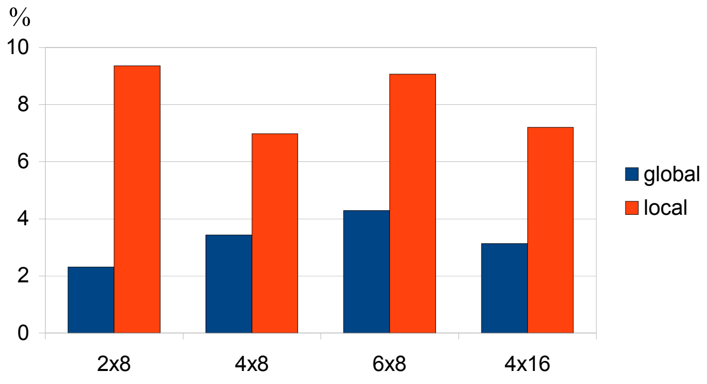


Figure 6.53 Task transfer percentages.

As a load distribution policy, the designed model is expected to reduce the unbalanced state of the cluster system. To measured the imbalace in load states, we calculated standard deviations of load values for both global and local load sharing processes. The results are shown in Figure 6.54. To compare the effect of the model on distribution of load, we also run the cluster submitting the same load pattern with load balancing policy disabled. The values shown as an example in the graph as "4x16nolb" are such results of a 4x16 cluster without load balancing. Comparing the results with those with load balancing enabled, it can be seen that the model reduces the load imbalance from about %19 to %9 locally and %7 to %2 in global load values. Although the results of other cluster types are generally close to each other, there are some slight differences in deviations. Use of nonpreemptive load transfers and dynamic nature of task submissions are main causes to these differences. Moreover, the threshold calculation method has a major effect on load levels. Since we used a threshold mechanism based on average distances, the load distribution policy tried to cause the load levels closer to the mean in some boundries (thresholds) instead of strictly equalizing them. Such a policy was avoided since it would have caused higher task transfer rates with extra overhead. The average distance values are shown in Figure 6.55. It can be seen that these results are closer to each other than those in standard deviations.

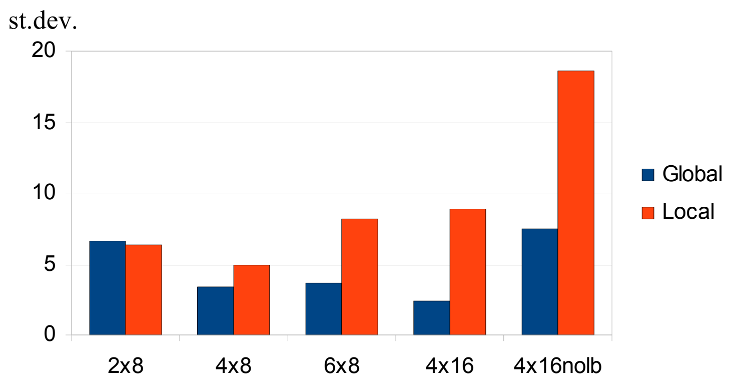


Figure 6.54 Standard deviation of load values.

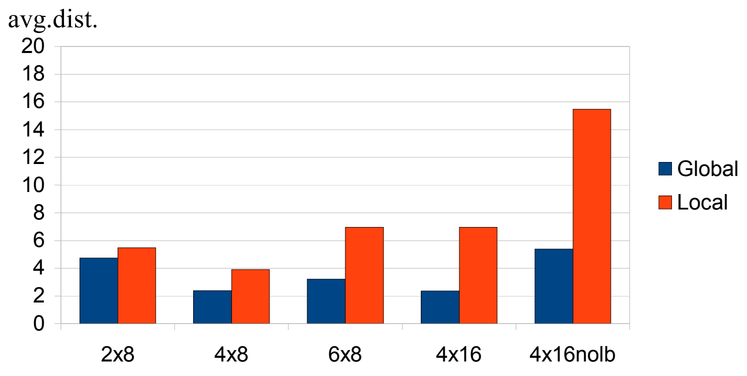


Figure 6.55 Average distances of load values.

Graphs plotting global load values and thresholds during runtime in Figure 6.56 show the effect of threshold mechanism more clearly. On the first graph that shows measurements on a 4x16 cluster without load balancing we see the spread of load values between %50 and %90. When load balancing enabled, values were shrunk towards the average and load interval narrowed to %60-80 around sender and receiver thresholds.

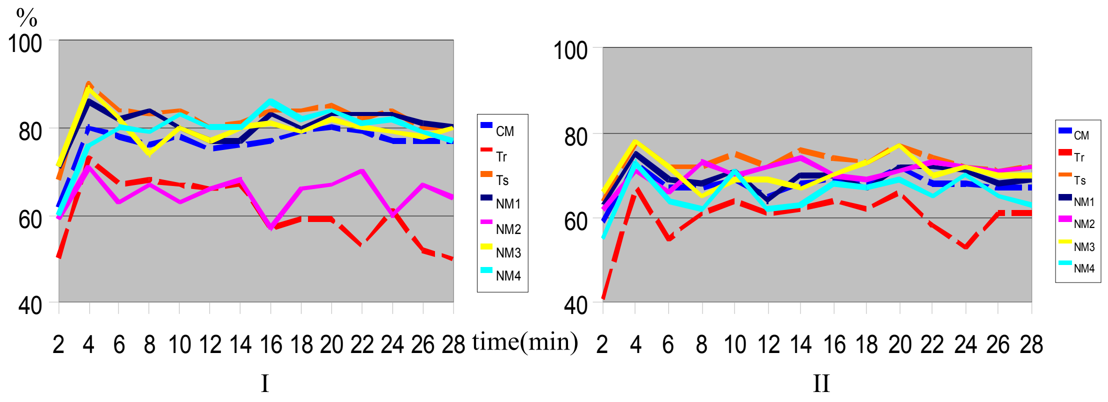


Figure 6.56 Global load values without (I) and with (II) global load sharing.

Finally, we show graphs of local load sharing policy for a 4x16 cluster in Figure 6.57. Graphs I and III shows the load levels of Ns for two groups without load balancing. Comparing these values to the corresponding measurements of those with load balancing enabled (graphs II and IV respectively) we see how the local load sharing shapes the load levels of Ns around the averages (shown by the dotted lines). With the help of load sharing and load index thresholds load levels of Ns with exceeding the limits were smoothed by task transfers through lightly loaded Ns.

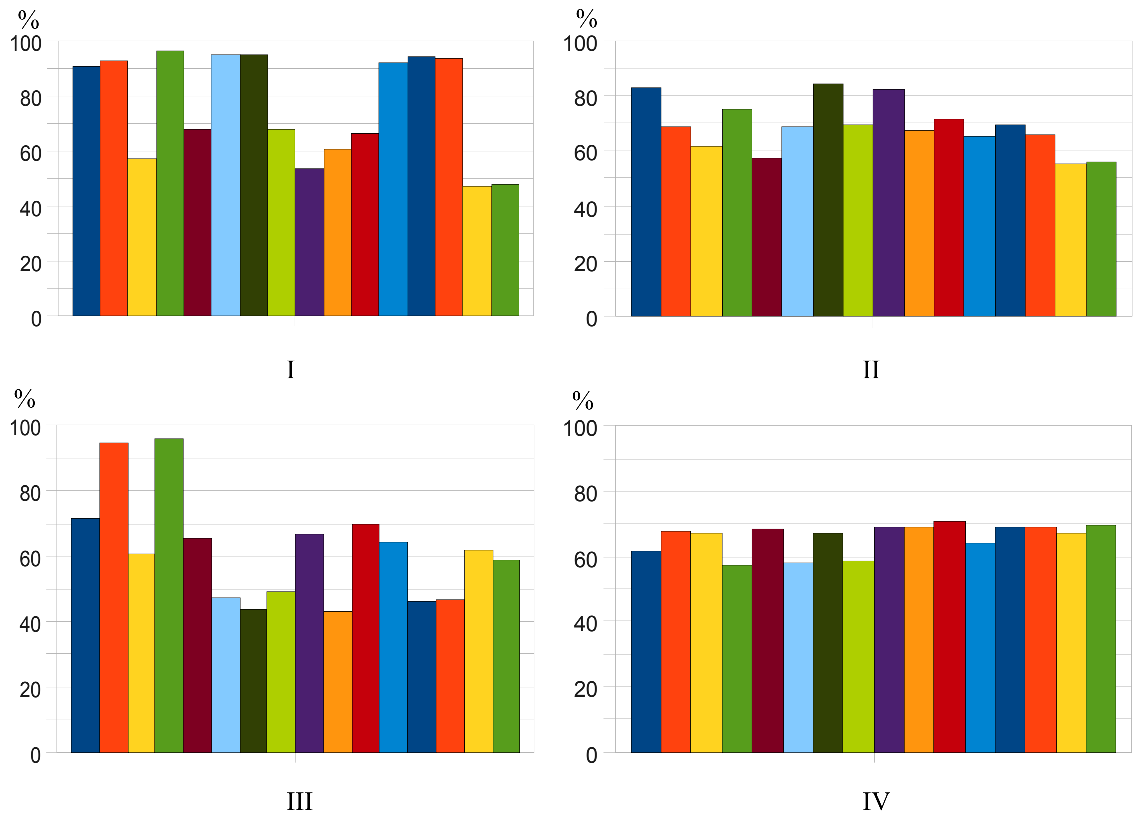


Figure 6.57 Local load values without (I,III) and with (II,IV) local load sharing.

# CHAPTER SEVEN
# CONCLUSIONS AND FUTURE DIRECTIONS

The research area of this thesis was about scalable Beowulf style clusters and efficient load distribution in these systems. A Beowulf cluster was defined as the cluster computing technology that connects tens to hundreds of personal computers together in such a way that they behave like a single computer which was a popular strategy for implementing parallel processing applications.

During researches it became clear that to build a stable and scalable cluster system an infrastructure that manages nodes of the cluster was needed. The Cluster Infrastructure Model (CIM) was designed for this purpose. In short, CIM is the foundation of the cluster system which is responsible for maintaining the components of the cluster by keeping the records of active nodes, checking their health and isolating failed nodes. Besides its this primary function, by its communication structure CIM also served an information service for the distributed load balancing model by collecting the state information from the nodes.

In researches it was seen that central policies could make more efficient load distribution decisions since the central controller had the complete knowledge about the whole cluster. However, they did not scale well on large clusters. On the other hand, distributed policies suffered from complexity and lack of complete knowledge to make most suitable decisions. To utilize the advantages of both techniques and avoid their disadvantages a hierarchically centralized architecture was designed.

To manage nodes of the cluster in a scalable manner, the system was hierarchically divided into a number of groups and these groups are under the control of node managers. On the upper level in the hierarchy there was a cluster manager as the leader of the node managers. CIM defined this structure with its communication and fault tolerance.

The cluster infrastructure model was implemented on Linux platform and tested on a simulation environment. The tests that were performed to evaluate the performance, stability and scalability of the model were joining a node to the cluster, removing a crashed node, replacing a dead node manager and cluster manager. Test results shown that the hierarchically layered model scales well on different size of clusters and its fault tolerance keeps the system stable in case of component failures.

The load balancing model was designed over the CIM. The hierarchical architecture of CIM also provided a scalable architecture for the load balancing model. By the hierarchical architecture parallel load distribution processes run at the lower level within groups and load distribution among them were organized on the upper layer. While local load distribution processes inside groups partially shared the loads inside groups, the global load distribution process completed the operation by distributing the loads among groups. The information needed for load distribution decisions are carried also via the health checking (heart-beat) messages of CIM by eliminating extra messaging requirement.

Support for heterogeneity of resources were provided by considering relative capacities for resources of the nodes. Moreover, the use of weighted multiple load indices that were taken into account to determine the load value of a node. This method provided a customizable and flexible model for a general purpose cluster. The load index thresholds were used to prevent resource overloading.

The dynamic sender and receiver threshold calculation method was used to add adaptivity property to the load balancing model. The threshold calculation method was designed as a customizable property.

Load index types, their weights, threshold calculation methods are customizable parts of the model that can be adjusted according to needs.

The load balancing model was implemented as a separate module integrated to the CIM. Some experimental tests were performed with different sizes of clusters to show the performance, efficiency and scalability of the designed model. The performed tests have shown how adaptive load threshold values shape the load levels around averages and avoid useless task transfers by successfully excluding moderately loaded members from source and destination selections.

There are some areas of possible future research, such as:

- Use of different threshold calculation methods to observe the changes in the sensitivity of the load distribution algorithm and also its overhead, e.g. using standard deviation instead of average distance.

- Testing of applications that require additional load index types other than CPU utilization and memory usage (like network usage, i/o queue length, etc.).

- Support for preemptive task transfers, that have been avoided so far because of their overhead and complexity, for applications where more aggressive load balancing is required.

- Implementing the model over a real cluster environment and testing it with a realistic problem.

## REFERENCES

Adams, D. A. (2005). *Optimal Load Balancing in a Beowulf Cluster.* MSc. Thesis in Computer Science, Worcester Polytechnic Institute.

Baker, M. & Buyya, R. (1999). *Cluster Computing At A Glance,* High Performance Cluster Computing, R. Buyya, Ed. Upper Saddle River, NJ: Prentice Hall PTR, vol. 1, Architectures and Systems, 3-47, chap. 1

Baker, M., & Buyya, R. (1988). Cluster Computing: The Commodity Supercomputing. *Software-Practice And Experience 1* (1), 1-4.

Casavant, T.L., Kuhl, J.G. (1988). A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering, vol. 14* (2), 141-154.

Castanegra, K., Cheng, D., & Fatoohi, R. (1994). *Clustered Workstations and their Potential Role as High Speed Compute Processors*. NAS Computational Services Technical Report, NAS Systems Division, NASA Ames Research Center.

Claypool, M., & Finkel, D. (2002). Transparent Process Migration For Distributed Applications in a Beowulf Cluster. *Proc. of the International Networking Conference*.

Dickson, K., Homic, C., & Villamin, S. B. (2000). *Putting PANTS On Linux: Transparent Load Sharing In A Beowulf Cluster*. Major Qualifying Project CS-DXF-9918.

Dongarra, J., Sterling, T., Simon, H., & Strohmaier, E. (2005). High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions. *Computing in Science and Engineering 7* (2), 51-59.

Harbaugh, L. G., (2004), *Building High-Performance Linux Clusters, Sponsored by Appro.* Retrieved September 2006, http://www.idgconnect.com/hardware/servers/building_high_performance_linux_clusters_sponsored_by_appro/

Hawick, K.A. , Grove, D.A., & Vaughan, F.A. (1999), Beowulf - a New Hope for Parallel Computing?. *DHPC Technical Report DHPC-061,* University of Adelaide

Hwang, K., & Xu, Z. (1998). *Scalable Parallel Computing: Technology, Architecture, Programming*. WCB/ McGraw-Hill, NY.

Kunz, T. (1991). The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme, *IEEE Trans. Software Eng., Vol. 17* (7) , 725-730.

Livny, M. & Melman, M. (1982). Load Balancing In Homogeneous Broadcast Distributed Systems, *Proc. ACM Computer Network Performance Symp. 11* (1), 47-55.

Meredith, M., Carrigan, T., Brockman, J., Cloninger, T., Privoznik, J., & Williams, J. (2003). Exploring Beowulf Clusters. *Journal of Computing Sciences in Colleges 18* (4), 268 – 284.

Shirazi, B. A., Husson, A. R., & Kavi., K. M. (1995). *Scheduling and Load Balancing in Parallel and Distributed Systems, chapter Introduction to Scheduling and Load Balancing*. IEEE Computer Society Press, Los Alamitos, CA, 1995.

Shivaratri, N. G., Krueger, P., & Singhal, M. (1992). Load Distributing For Locally Distributed Systems. *IEEE Computer, 25*, 33-44.

Soria, M., Pérez-Segarra, C. D., & Oliva, A. (2002). A Direct Parallel Algorithm For The Efficient Solution Of The Pressure-Correction Equation Of Incompressible Flow Problems Using Loosely Coupled Computers. *Numerical Heat Transfer, Part B: Fundamentals 41* (2), 117-138.

Sterling, T., Becker, D. J., Dorband, J. E., Savarese, D., Ranawake, U. A., & Packer, C. V. (1995). A Parallel Workstation for Scientific Computation, *Proc. of the 24th International Conference on Parallel Processing*.

Sunderam, V., Geist, G., Dongarra, J., & Manchek, R. (1994). The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing, 20* (4).

Thain, D., Tannenbaum, T., & Livny, M. (2005). Distributed Computing in Practice: The Condor Experience, *Concurrency and Computation: Practice and Experience 17* (2-4), 323-356.

Walker, D., & Dongarra, J. (1996). MPI: A Standard Message Passing Interface. *Supercomputing, 12* (1).

# APPENDICES

## Appendix A - Pseudocodes of CIM Modules

### A.1 Pseudocode of CM Module

```
Program ClusterManager
{
 Initialize_variables
 sendmsg(ClusterMulticastAddr,NewCM)
 msg_count=1
 reply_count=0
 start_timer(Startup)
 start_timer(Activity)
 state=Startup

 Loop Forever
 {
   wait_event
   Case (event)
   Startup_timeout:
        if msg_count<max_NewCM_msg_sent_count
          sendmsg(ClusterMulticastAddr,NewCM)
             msg_count=msg_count+1
             start_timer(Startup)
        else
             state=Ready
   NMOK_msg_received:
        set_NM_info
        if reply_count=NMCount
             stop_timer(Startup)
             state=Ready
   Heartbeat_msg_received:
        active[sender_NM]=1
        if state=CrashDetect
             stop_timer(CrashDetect)
             state=Ready
   Heartbeat_timeout:
        if state=Ready
             sendmsg(timedout_NM,CheckAlive)
             start_timer(CheckAlive)
        state=CrashDetect
   else
        postpone_event
   CheckAlive_timeout:
        remove(timeout_NM)
```

```
          sendmsg(backupN,StartNM)
          state=Ready
     CheckAlive_msg_received:
          sendmsg(sender_NM,Heartbeat)
     Ncrash_msg_received:
          if state=Ready or Split
               remove(crashedN)
          else
               postpone_event
     NJoin_msg_received:
          if state=Ready
               sendmsg(selectedNM,AddN)
          else
               postpone_event
     NAdded_msg_received:
          if state=Ready
               Add(N)
          else
               postpone_event
     Split_condition_occurred:
          if state=Ready
               sendmsg(ClusterMulticast_addr,Split)
               start_timer(Split)
               msg_count=1
               state=Split
     Split_timeout:
          if msg_count<max_Split_msg_sent_count
            sendmsg(ClusterMulticastAddr,Split)
               msg_count=msg_count+1
               start_timer(Split)
          else
               state=Ready
     Activity_timeout:
          sendmsg(backup_NM,Heartbeat)
     End Case
   }
```

## A.2 Pseudocode of NM Module

```
  Program NodeManager
  {
   Initialize_variables
   sendmsg(GroupMulticastAddr,NewNM)
   msg_count=1
   reply_count=0
   start_timer(Startup)
   start_timer(Activity)
   state=Startup
```

```
Loop Forever
{
  wait_event
  Case (event)
  Startup_timeout:
       if msg_count<max_NewNM_msg_sent_count
         sendmsg(ClusterMulticastAddr,NewNM)
            msg_count=msg_count+1
            start_timer(Startup)
       else
            state=Ready
  NOK_msg_received:
       if state=Startup
            set_N_info
            if reply_count=NCount
                 stop_timer(Startup)
                 state=Ready
       if state=NewN
            stop_timer(NewN)
            Add(N)
            sendmsg(CM,NAdded)
            state=Ready
       if state=Split
            if NOK_msg_count=split_count
            stop_timer(NMUpdate)
                 sendmsg(CM,NMOK)
                 state=Read
  Heartbeat_msg_received:
       active[sender_N]=1
       if state=NCrashDetect or CMCrashDetect
            stop_timer(CrashDetect)
            state=Ready
  Heartbeat_timeout:
       if state=Ready
            if N_timedout
                 sendmsg(timedout_N,CheckAlive)
            state=NCrashDetect
            if CM_timedout
                 sendmsg(CM,CheckAlive)
            state=CMCrashDetect
            start_timer(CheckAlive)
  else
       postpone_event
  CheckAlive_timeout:
       if state=NCrashDetect
            remove(crashed_N)
            sendmsg(CM,NCrash)
       if state=CMCrashDetect
```

```
                StartCM
           state=Ready
      CheckAlive_msg_received:
           sendmsg(sender_N or CM,Heartbeat)
      AddN_msg_received:
           if state=Ready
                sendmsg(new_N,NAccepted)
                start_timer(NewN)
                state=NewN
           else
                postpone_event
      NewN_timeout:
           state=Ready
      NewCM_msg_received:
           if state=Ready
                sendmsg(CM,NMOK)
           else
                postpone_event
      Split_msg_received:
           if state=Ready
                sendmsg(GroupMulticast_addr,NMUpdate)
                start_timer(NMUpdate)
                msg_count=1
                state=Split
           else
                postpone_event
      NMUpdate_timeout:
           if msg_count<max_NMUpdate_msg_sent_count
              sendmsg(GroupMulticastAddr,Split)
                msg_count=msg_count+1
                start_timer(NMUpdate)
           else
                sendmsg(CM,NCrash)
                sendmsg(CM,NMOK)
                state=Ready
      Activity_timeout:
           sendmsg(timedout_N or CM,Heartbeat)
      End Case
    }
```

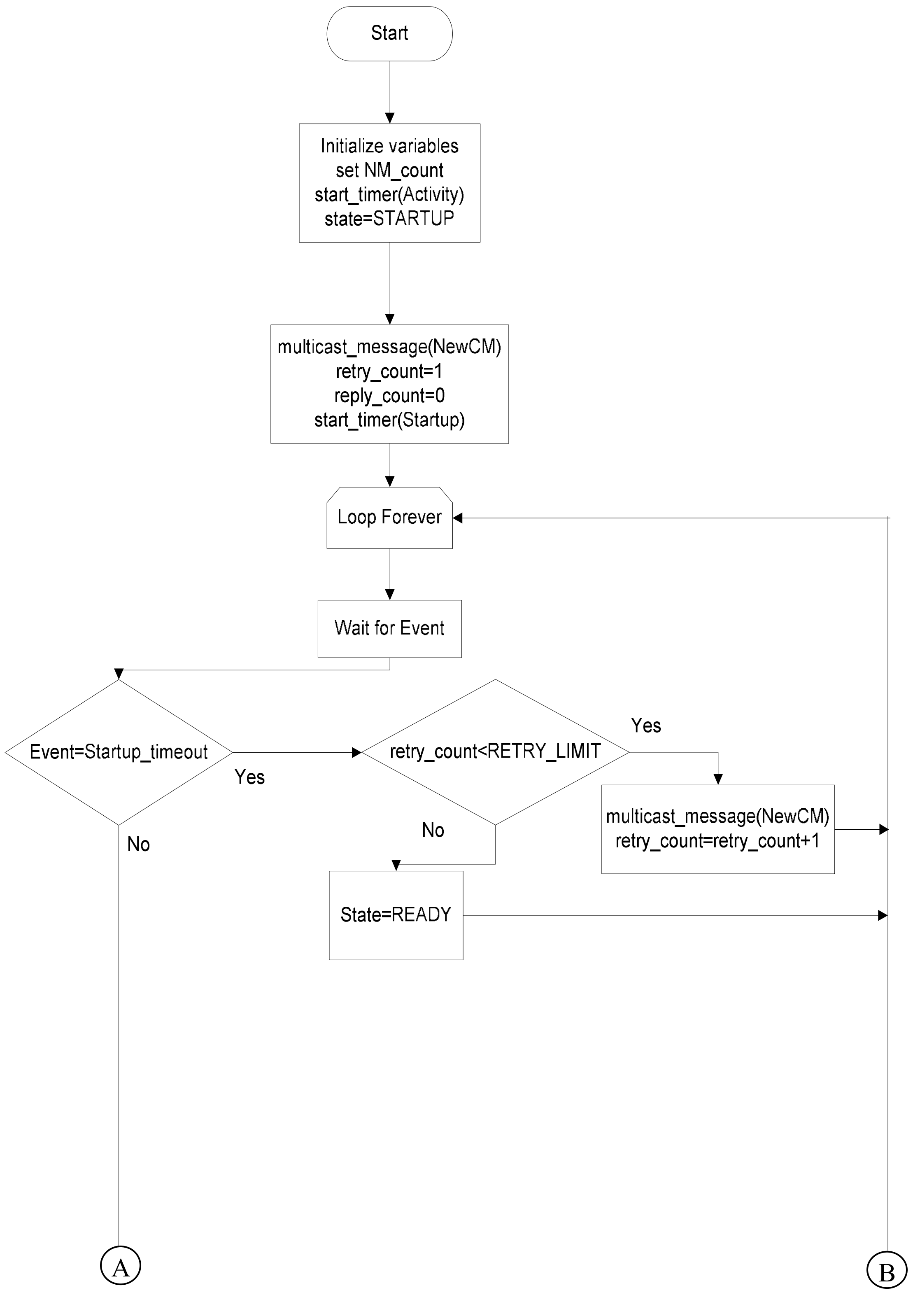
**A.3 Pseudocode of N Module**

```
Program Node
{
 Initialize_variables
 sendmsg(ClusterMulticastAddr,NJoin)
 start_timer(Startup)
 start_timer(Activity)
```

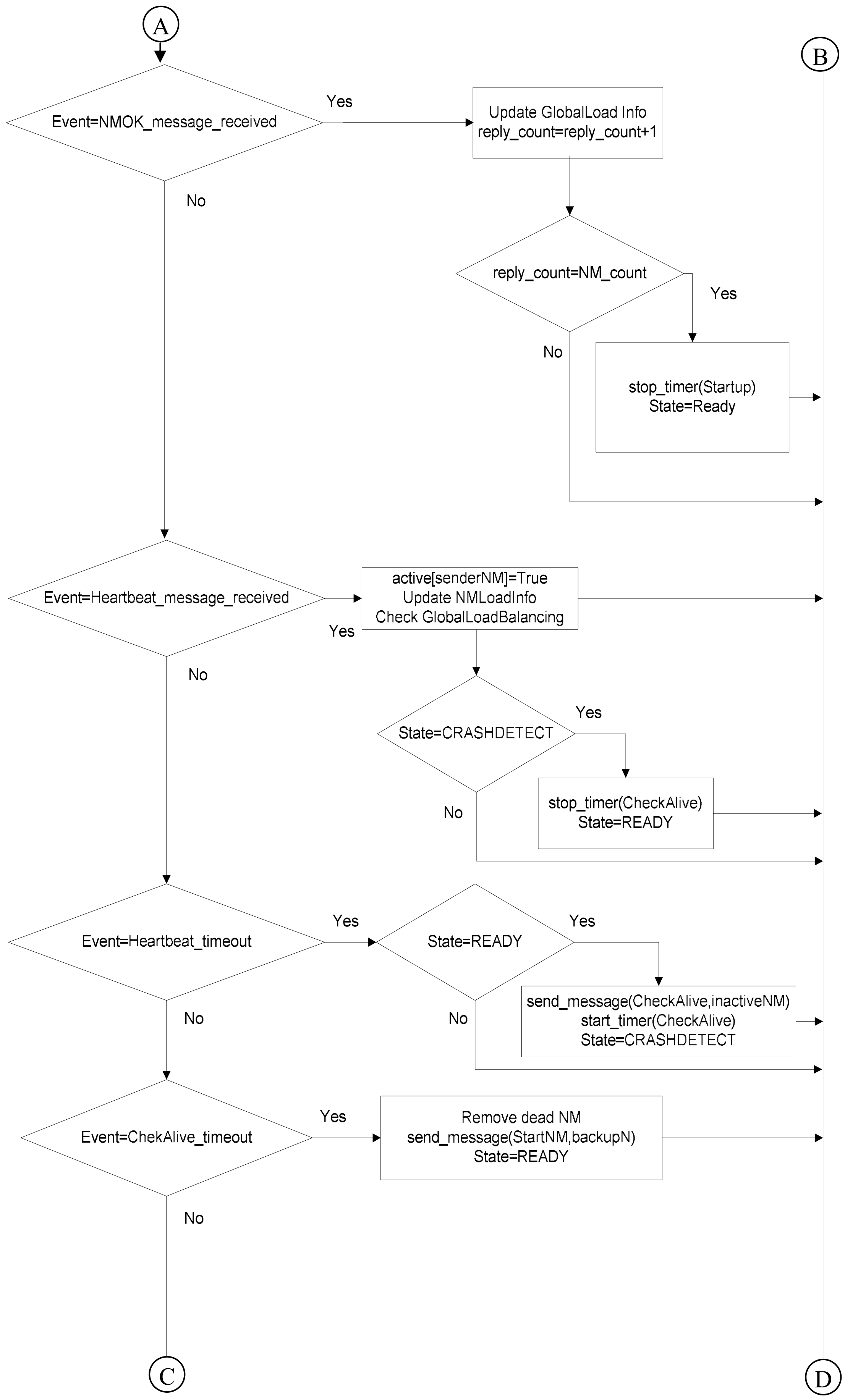
```
state=Startup

Loop Forever
{
  wait_event
  Case (event)
  Startup_timeout:
    sendmsg(CMMulticastAddr,NJoin)
       start_timer(Startup)
  NAccepted_msg_received:
       stop_timer(Startup)
       set_group_info
       sendmsg(NM,NOK)
    start_timer(Activity)
       state=Ready
  CheckAlive_msg_received:
       sendmsg(NM,Heartbeat)
  NMUpdate_msg_received:
       sendmsg(NM,NOK)
       Update_NM_info
  NewNM_msg_received:
       Update_group_info
       sendmsg(NM,NOK)
  Activity_timeout:
       sendmsg(NM,Heartbeat)
  End Case
}
```

# Appendix B - Flowcharts of Load Balancing Modules

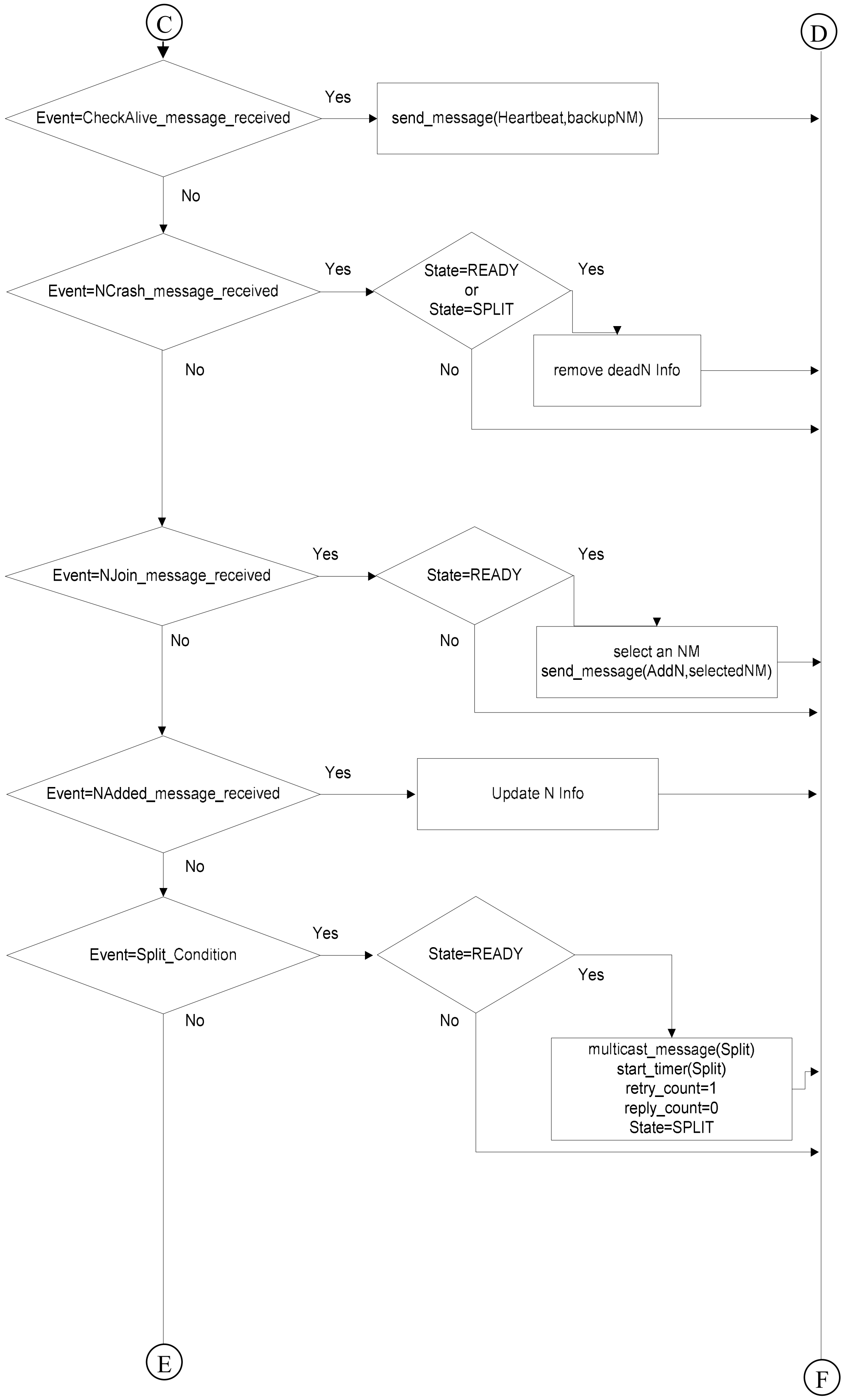
## B.1 Flowchart of CM Module



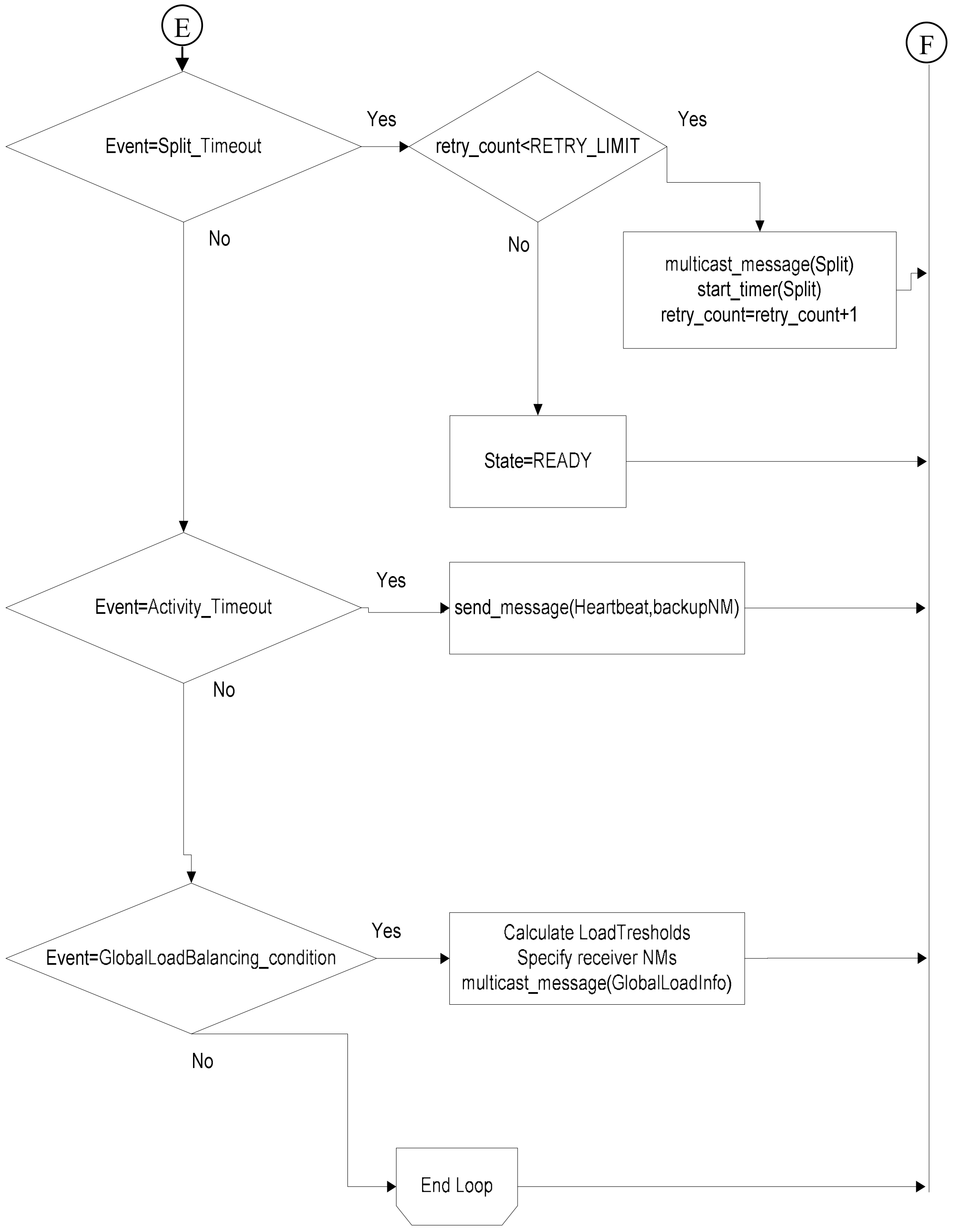
## B.1 Flowchart of CM Module (continued)



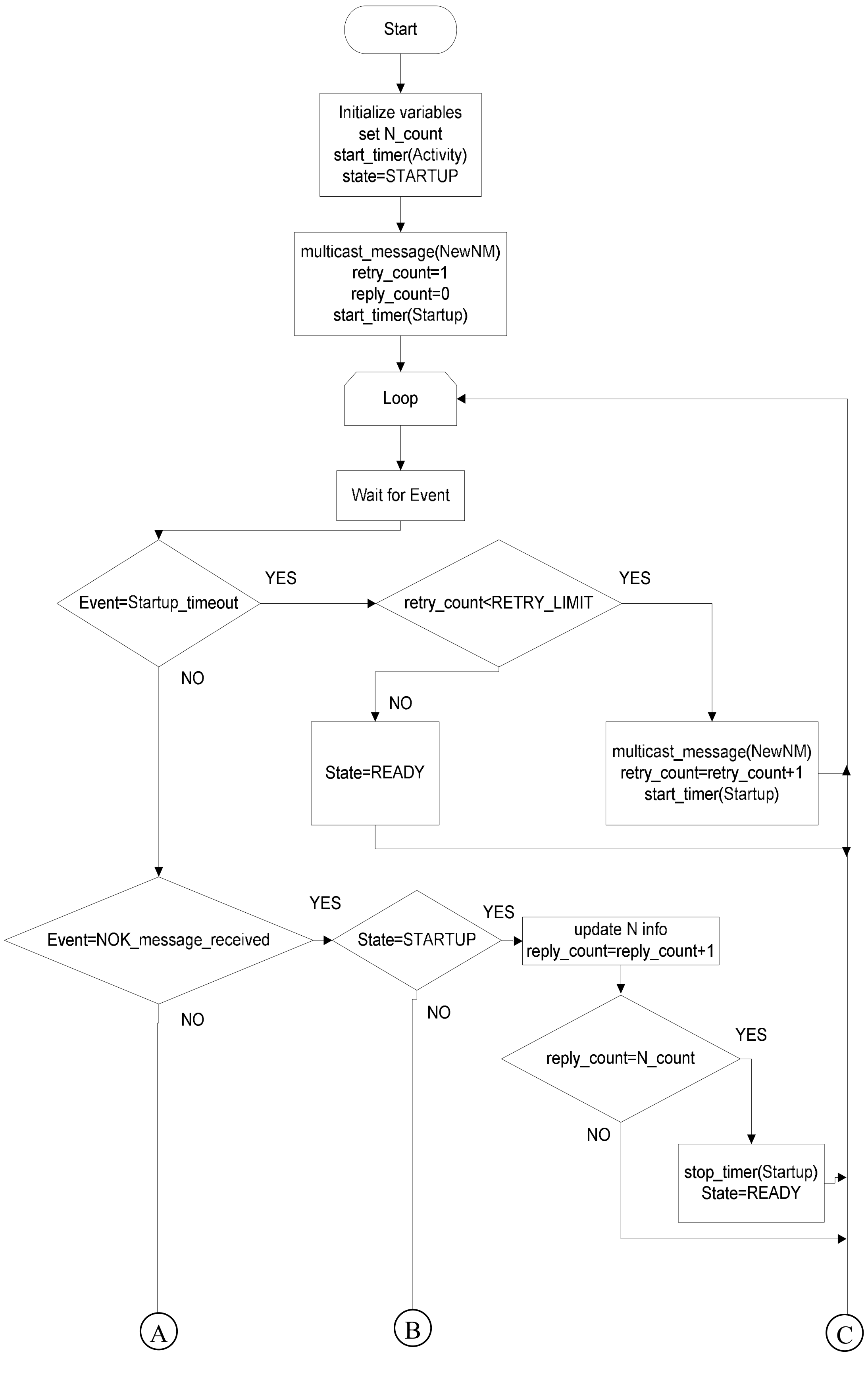
## B.1 Flowchart of CM Module (continued)


C
D
Event=CheckAlive_message_received
Yes
send_message(Heartbeat,backupNM)
No
Event=NCrash_message_received
Yes
State=READY or State=SPLIT
Yes
remove deadN Info
No
No
Event=NJoin_message_received
Yes
State=READY
Yes
select an NM send_message(AddN,selectedNM)
No
No
Event=NAdded_message_received
Yes
Update N Info
No
Event=Split_Condition
Yes
State=READY
Yes
multicast_message(Split) start_timer(Split) retry_count=1 reply_count=0 State=SPLIT
No
No
E
F

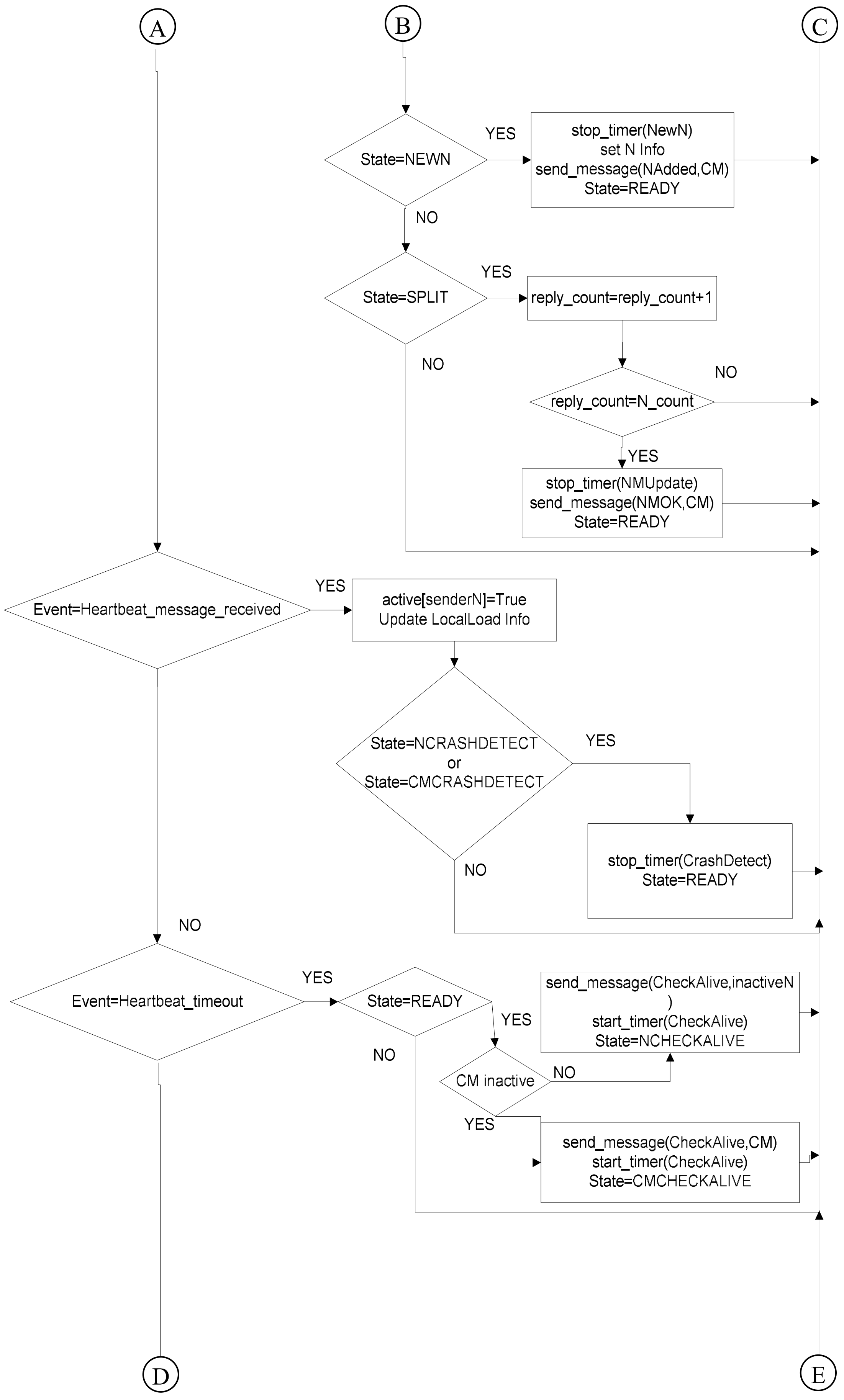
## B.1 Flowchart of CM Module (continued)



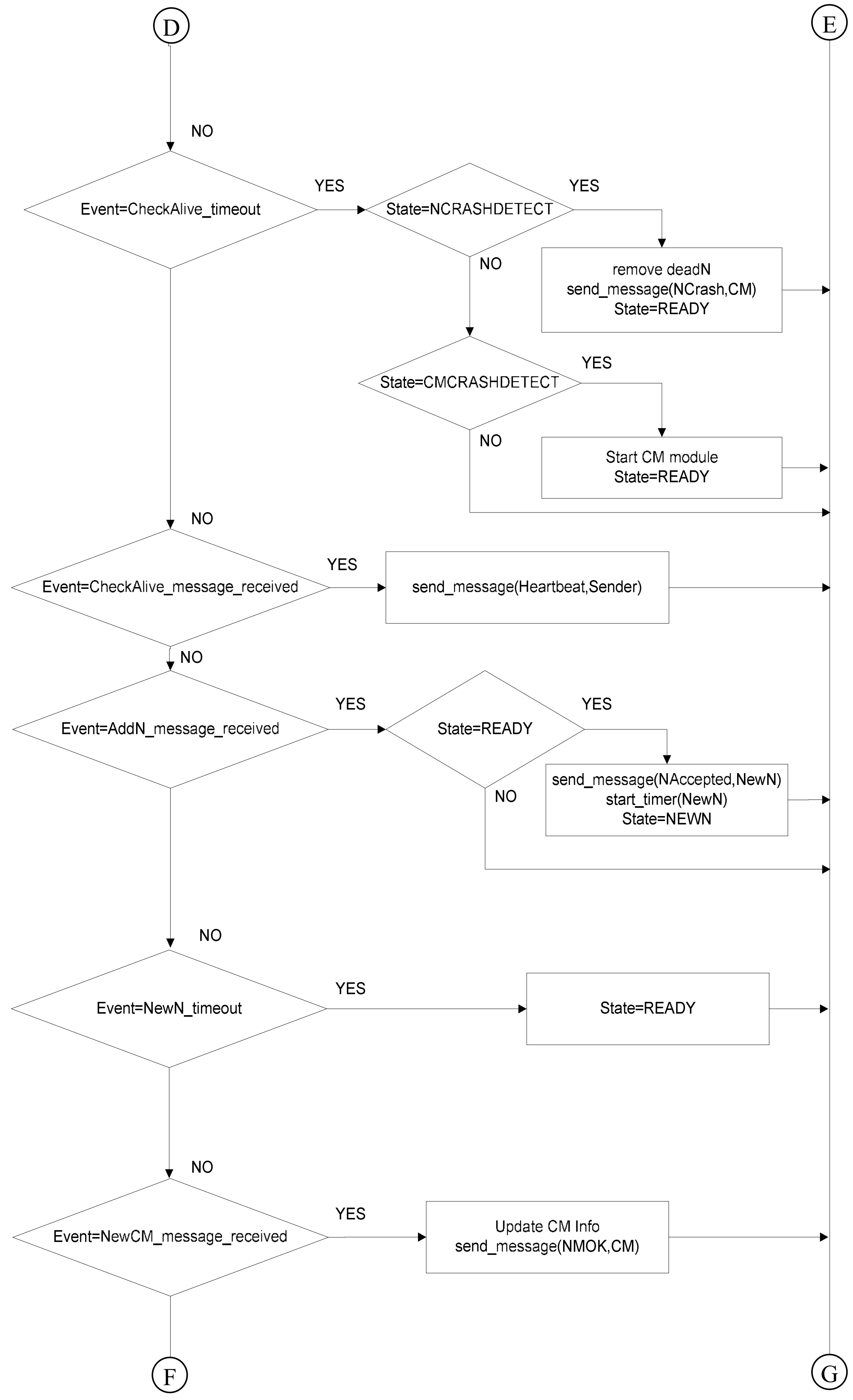
## B.2 Flowchart of NM Module



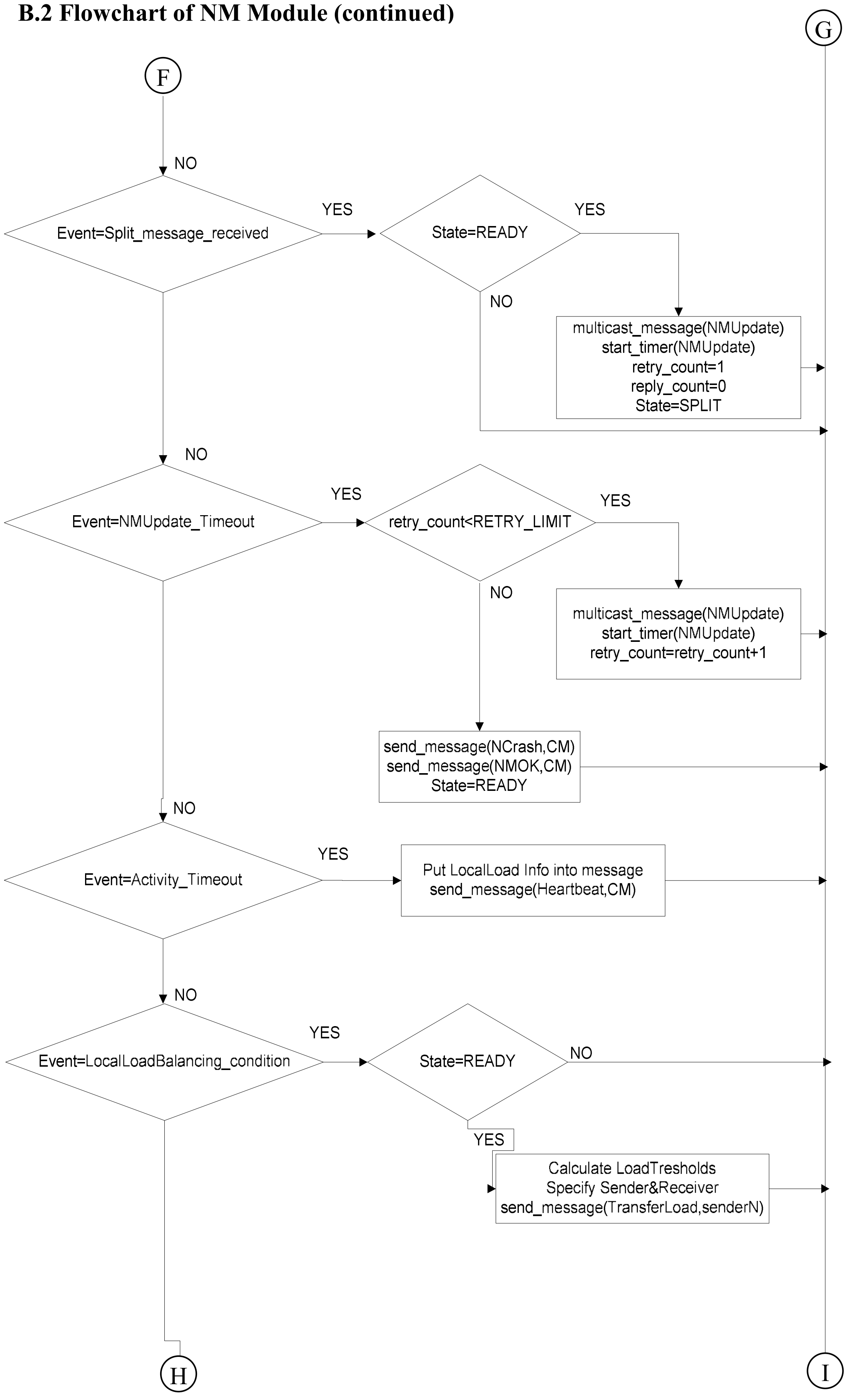
## B.2 Flowchart of NM Module (continued)



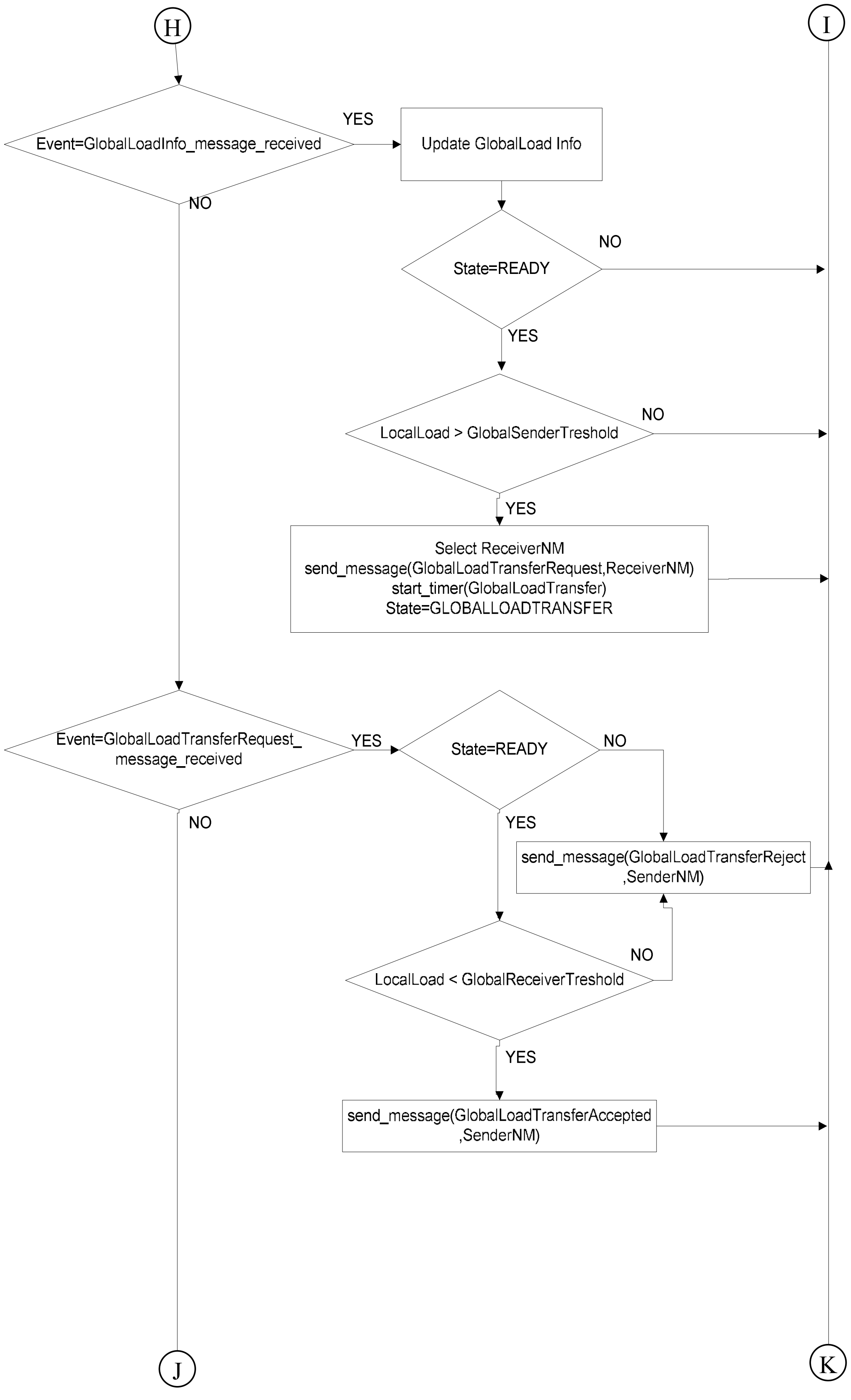
## B.2 Flowchart of NM Module (continued)



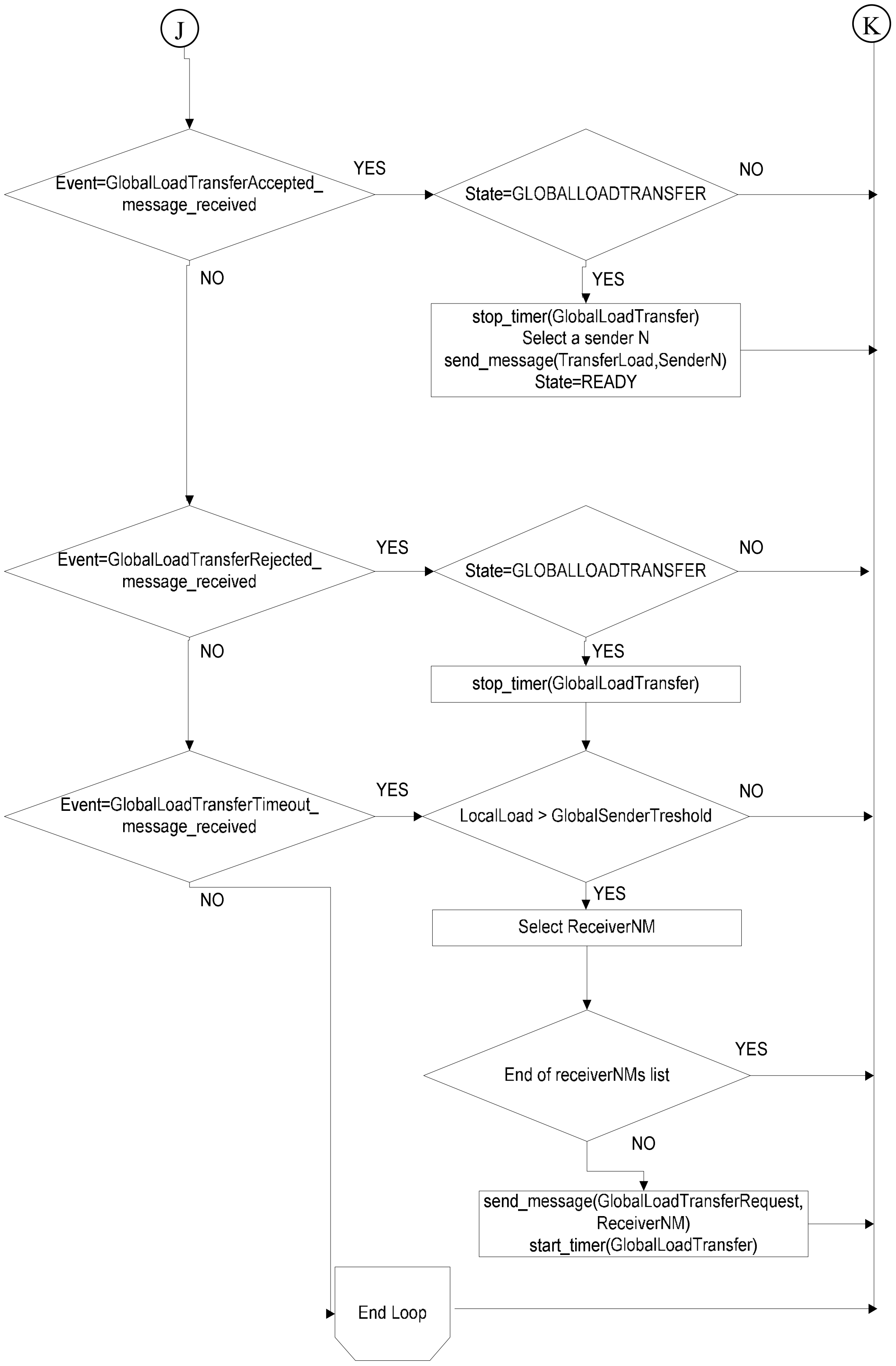
## B.2 Flowchart of NM Module (continued)



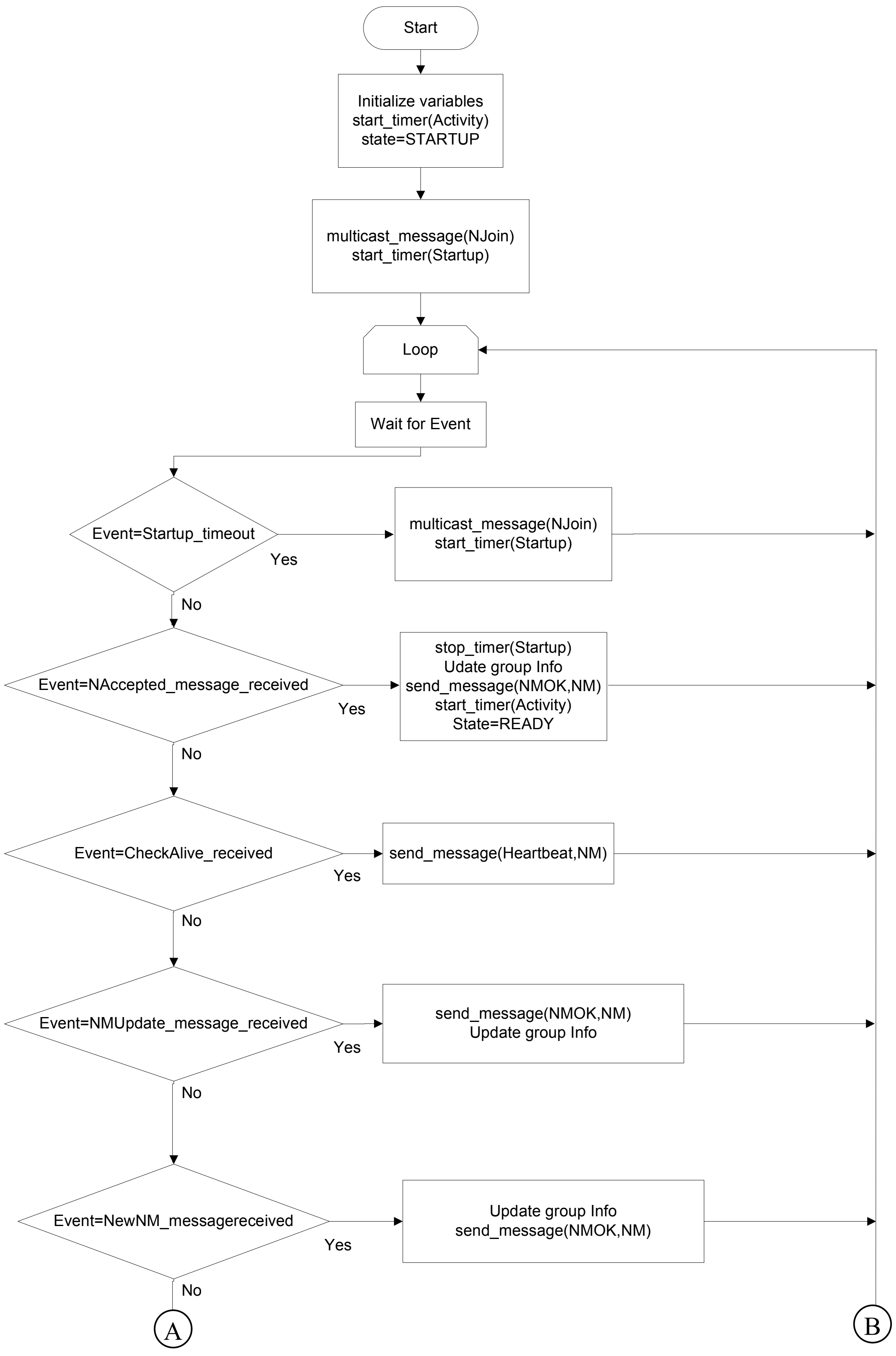
## B.2 Flowchart of NM Module (continued)


H
I
Event=GlobalLoadInfo_message_received
YES
NO
Update GlobalLoad Info
State=READY
NO
YES
LocalLoad > GlobalSenderTreshold
NO
YES
Select ReceiverNM
send_message(GlobalLoadTransferRequest,ReceiverNM)
start_timer(GlobalLoadTransfer)
State=GLOBALLOADTRANSFER
Event=GlobalLoadTransferRequest_ message_received
YES
NO
State=READY
NO
YES
send_message(GlobalLoadTransferReject ,SenderNM)
LocalLoad < GlobalReceiverTreshold
NO
YES
send_message(GlobalLoadTransferAccepted ,SenderNM)
J
K

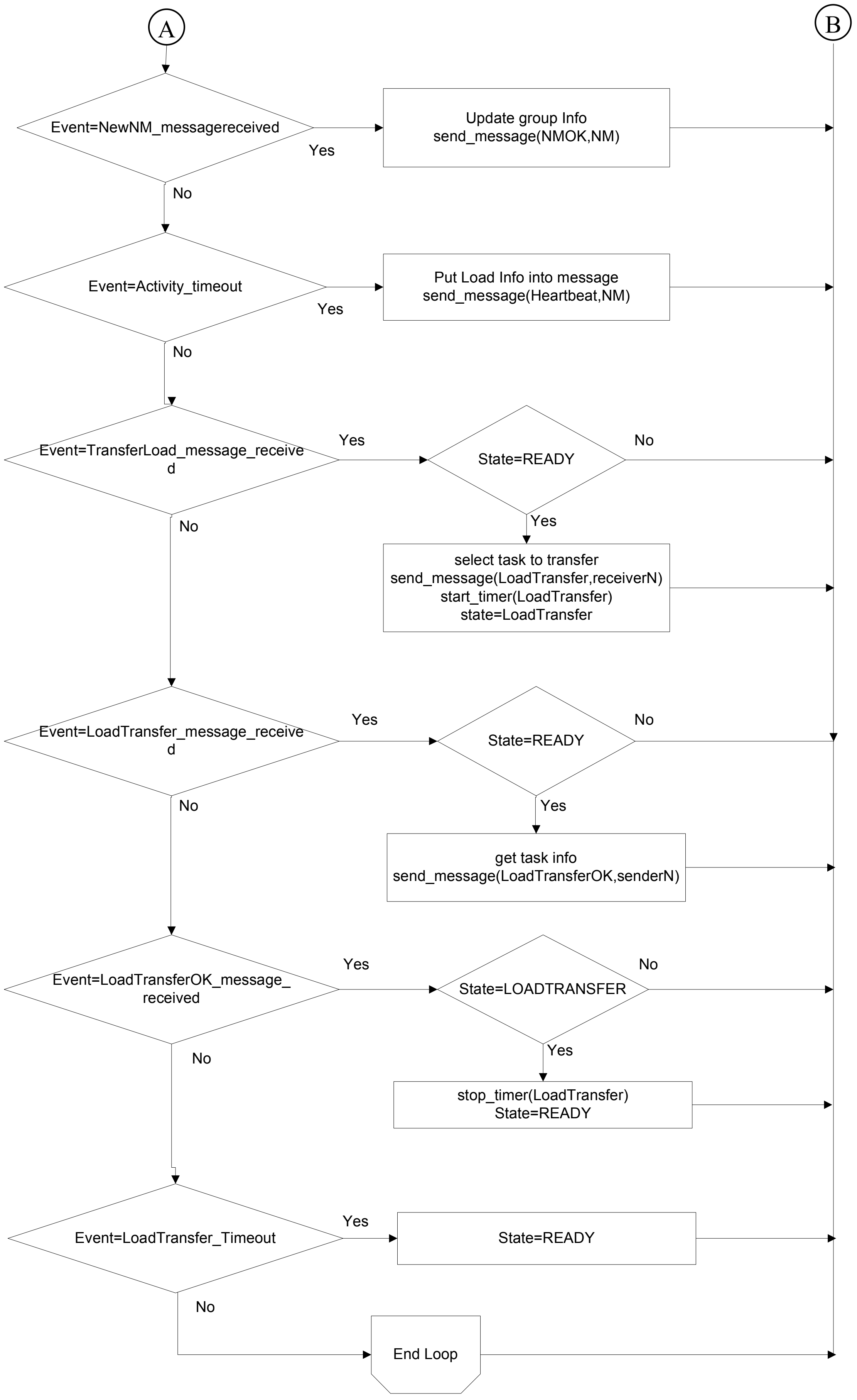
## B.2 Flowchart of NM Module (continued)


J
K
Event=GlobalLoadTransferAccepted_ message_received
YES
NO
State=GLOBALLOADTRANSFER
NO
YES
stop_timer(GlobalLoadTransfer)
Select a sender N
send_message(TransferLoad,SenderN)
State=READY
Event=GlobalLoadTransferRejected_ message_received
YES
NO
State=GLOBALLOADTRANSFER
NO
YES
stop_timer(GlobalLoadTransfer)
Event=GlobalLoadTransferTimeout_ message_received
YES
NO
LocalLoad > GlobalSenderTreshold
NO
YES
Select ReceiverNM
End of receiverNMs list
YES
NO
send_message(GlobalLoadTransferRequest, ReceiverNM)
start_timer(GlobalLoadTransfer)
End Loop

## B.3 Flowchart of N Module



## B.3 Flowchart of N Module (continued)


A
B
Event=NewNM_messagereceived
Yes
No
Update group Info
send_message(NMOK,NM)
Event=Activity_timeout
Yes
No
Put Load Info into message
send_message(Heartbeat,NM)
Event=TransferLoad_message_received
Yes
No
State=READY
No
Yes
select task to transfer
send_message(LoadTransfer,receiverN)
start_timer(LoadTransfer)
state=LoadTransfer
Event=LoadTransfer_message_received
Yes
No
State=READY
No
Yes
get task info
send_message(LoadTransferOK,senderN)
Event=LoadTransferOK_message_received
Yes
No
State=LOADTRANSFER
No
Yes
stop_timer(LoadTransfer)
State=READY
Event=LoadTransfer_Timeout
Yes
No
State=READY
End Loop