Implementation of Load Balancing Policies in Distributed Systems

by

Jean Ghanem

B.E., American University of Beirut, 2002

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Electrical Engineering

The University of New Mexico

Albuquerque, New Mexico

June, 2004

Dedication

To my dearest parents and my beautiful fiancée.

Acknowledgments

I would like to dedicate my thesis to my parents who always stood beside me in everything I did. Thank you mother for been always so loving and caring. Thank you father for making me always strive for better. Thank you Sabine and Samer for being the best friends whom I can rely on.

I would also like to dedicate my thesis to the Hamadé family who were always there in my good times and bad times. Thank you for always believing in me and for your continuous support.

I would also like to dedicate my thesis[1] to my advisor and mentor Professor Chaouki Abdallah who was the source of my motivation and inspiration, through his continuous guidance, encouragement and patience. Thank you Professor for everything, I will be forever grateful.

I would like to thank Professor Majeed Hayat for his expertise in the field of load balancing, for all the valuable discussions that we had and for his continuous support. I would also like to thank my committee member Professor Gregory Heileman for his helpful comments.

I would like to express my sincere gratitude to Mr. Henry Jerez for his help in this thesis work and for sharing his great knowledge in the field of networking and distributed systems. I would also like to thank my colleague Mr. Sagar Dhakal for his great help in this thesis work. It was a pleasure working with you.

Last but not least, I would like to dedicate this work and extend my warmest gratitude to my beautiful fiancée Nayla. Thank you Nayla for correcting and enhancing my thesis. Thank you for being the person that gave me the strength to survive throughout all my bad moments. Thank you for being the trusting and confiding person I relied on during our stay in Albuquerque. Thank you my love for being the sincere and wonderful person to whom I will devote my entire life.

[1]This work was supported by the National Science Foundation under Information Technology Research (ITR) grant No. ANI-0312611 and ANI-0312182

Implementation of Load Balancing Policies in Distributed Systems

by

Jean Ghanem

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science
Electrical Engineering

The University of New Mexico

Albuquerque, New Mexico

June, 2004

Implementation of Load Balancing Policies in Distributed Systems

by

Jean Ghanem

B.E., American University of Beirut, 2002

M.S., Electrical Engineering, University of New Mexico, 2004

Abstract

Load balancing is the allocation of the workload among a set of co-operating computational elements (CEs). In large-scale distributed computing systems, in which the CEs are physically or virtually distant from each other, there are communication-related delays that can significantly alter the expected performance of load-balancing policies that do not account for such delays. This is a particularly prominent problem in systems for which the individual units are connected by means of a shared communication medium such as the Internet, ad-hoc networks, wireless LANs. Moreover, the system performance may greatly vary since it incorporates heterogenous nodes that are not necessarily dedicated to the application at hand. In such cases, an actual implementation becomes necessary to understand the load-balancing strategies and their reactions when employed in several environments since mathematical models may not always capture the unpredictable behavior of such systems.

In this thesis we propose a software implementation architecture where several distributed load-balancing strategies could be tested and verified under different environments. We then experimentally investigate network delays that are the main factor in degrading the performance of the load distribution strategies. Subsequently, we test the different policies on our test-bed and use the results to develop an improved policy that adapts to the system parameters such as transfer delays, connectivity, and CE computational power.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Description and Motivation

The demand for high performance computing continues to increase everyday. The computational need in areas like cosmology, molecular biology, nanomaterials, etc., cannot be met even by the fastest computers available [6, 29]. But with the availability of high speed networks, a large number of geographically distributed computational elements (CEs) can be interconnected and effectively utilized in order to achieve performances not ordinarily attainable on a single CE. The distributed nature of this type of computing environment calls for consideration of heterogeneities in computational and communication resources. A common architecture is the cluster of otherwise independent CEs communicating through a shared network. An incoming workload has to be efficiently allocated to these CEs so that no single CE is overburdened, while one or more other CEs remain idle. Further, tasks migration from high to low traffic area in a network may alleviate to some extent the network-traffic congestion problem.

Workstation clusters are being recognized as the most promising computing resource of the near future. A large-size cluster, consisting of locally connected workstations, has power comparable to a supercomputer, at a fraction of the cost. Furthermore, a wide-area coupling of workstation clusters is not only suitable for exchange of mail and news or the establishment of distributed information systems, but can also be exploited as a large metacomputer [9]. In theory, a metacomputer is a similarly easy-to-use assembly of distinct computers or processors working together to tackle a single task or a set of problems. Distributing the total computational load across available processors is referred to in the literature as *load-balancing*.

Effective load-balancing of a cluster of CEs in a distributed computing system relies on accurate knowledge of the state of the individual CEs. This knowledge is used to judiciously assign incoming computational tasks to appropriate CEs, according to some load-balancing policy. In large-scale distributed computing systems in which the CEs are physically or virtually distant from each other, there are a number of inherent time-delay factors that can seriously alter the expected performance of load-balancing policies that do not account for such delays. One manifestation of such time delay is attributable to the computational limitations of individual CEs. A more significant manifestation of such delay arises from the communication limitations between the CEs. These include delays in transferring loads amongst CEs and delays in the communication between them. Moreover, such delays not only fluctuate within each CE as the amounts of the loads to be transferred vary, but also vary as a result of the uncertainties in the communication medium that connects the units. This kind of delay-uncertainty is frequently observed in systems for which the individual units are connected by means of a shared communication medium (e.g., the Internet, ATM, ad-hoc networks, wireless LANs) [24].

There has been extensive research in the development of dynamic load-balancing policies. Some of these existing approaches assume constant performance of the network while others assume deterministic communication and transfer delay. The load-balancing schemes designed under such assumptions ignore randomness in delay [18, 20]. In this thesis, we propose a software implementation of a general framework where distributed and dynamic load-balancing policies could be tested. We then test them in different environment namely the Internet and wireless networks to finally come up with an improved adaptive policy.

To adequately model load-balancing problems, several features of the parallel computation environment should be captured: These include (1) The workload awaiting processing at each CE (i.e., queue size); (2) The relative performances of the CEs; (3) The computational requirements of each workload component; (4) The delays and bandwidth constraints of CEs and network components involved in the exchange of workloads, and (5) The delays imposed by CEs and the network on the exchange of measurements and information [24, 22]. The effect of delay in particular is expected to be a key factor as searching large databases moves toward distributed architectures with potentially geographically distant units.

1.2 General Framework for load-balancing

A typical distributed system will have a number of processors working independently with each other. Some of them are linked by communication channel and while some are not. Each processor possesses an initial load, which represents an amount of work to be performed, and each may have a different processing capacity. To minimize the time needed to perform all tasks, the workload has to be evenly distributed over all processors based on their processing speed. This is why load-balancing is needed. If all communication links are infinite bandwidth, the load distribution would suffer from no delay, but this does not represent real distributed environments. In any practical distributed systems, the channels are of finite bandwidth and the processing units may be physically distant; Therefore, load-balancing is also a decision making process of whether to allow tasks migration or not.

Another issue related to load-balancing is that a computing job may not arbitrarily divisible leading to certain constraints in dividing tasks. Each job consists of several smaller tasks and each of those tasks can have different execution times. Also, the load on each processor as well as on the network can vary from time to time based on the workload brought about by the users. The processor capacity may be different from each other in architecture, operation system, CPU speed, memory size, and available disk space. The load-balancing problem also needs to consider fault-tolerance and fault-recovery. With all these factors taken into account, load-balancing can be generalized into four basic steps: (1) Monitoring processor load and state; (2) Exchanging load and state information between processors; (3) Calculating the new work distribution; and (4) Actual data movement. In this scheme, numerous load-balancing strategies are available but they all could be implemented on the same test-bed since they share the same basic steps described above.

1.3 Objective of this Thesis

The main goal of this thesis is to experimentally investigate the behavior of distributed load-balancing policies in a real environment. Analytical and queueing models may not always take into account all the parameters and inputs of an actual system that has unpredictable behavior. Therefore, it is crucial to experiment the aspects of the policies under actual conditions to check the system's response and come up with *heuristic* improvements. Hence, we propose a software implementation of a load-balancing system where we examine how its three components, application, load distribution, and network communication should interact to provide high throughput to the application at hand regardless of the policy adopted. Moreover, to better understand the reactions of the policies in large networks, we conduct an experimental analysis of the network delays and categorize them according to their characteristics. After investigating the basic policies and the effect of delays on the stability of the systems they act upon, we propose adaptive load-balancing policies that account for several system parameters including CE computational power and interconnection delays.

For a given workload distribution among a group of heterogeneous processors, we recognize the overall completion time of the group [20, 19] and the stability point where the load is evenly distributed across the system as the performance metrics [22]. The words "stable" and "stability" acquired from the controls area will be used throughout the thesis (especially in Chapter 5) to denote the evenness degree of the load distribution across the processors. The objective is to develop a balancing strategy which minimizes both of these parameters. We do not address the process selection [33] for migration since we assume that each task has a priority attached to it as indicated by its insertion order to the queue of each node.

This thesis may also have the potential for being useful in other fields such as networked control systems (NCS) and teleautonomy. In a NCS the sensor and the controller are connected over a shared network and therefore, there is a delay in closing the feedback loop. A special application of teleautonomy [37, 38] is that of robots distributed geographically and working autonomously but at the same time being monitored by a distant controller. Clearly, load distribution may be needed across the robots where communication delays may degrade the performance of such systems.

1.4 Overview of Thesis

In Chapter 2, we present an overview of existing load-balancing strategies. We start by briefly discussing different schemes. We then look into special types of load-balancing schemes available in the literature and the queueing models on which the policies adopted in this thesis are based. Chapter 3 presents the internal software architecture of the proposed test-bed system followed by a description of the policies that were integrated in it. Chapter 4 introduces delay probing experiments performed on the Internet and classified according to their variability. We then conduct delay experiments on the wireless network whose results are integrated in the Monte-Carlo load-balancing simulator of the stochastic queueing model presented in Section 2.3.2. In Chapter 5, experimental results conducted on the implemented policies over two different test-beds, the Internet and a wireless network are presented. The effect of delays and the variation in the CEs performance are examined to see how they influence the system's ability to reach a load balanced state. Finally, based on previous observations regarding the behavior of network delays and the performance of the policies in distributed systems, we propose in Chapter 6 a dynamic and adaptive load-balancing policy that accounts for such parameters. Chapter 7 presents our conclusions and suggestions for future research.

Chapter 2

Load-Balancing Taxonomy and Previous Work

In this chapter, a brief overview of the different taxonomies of load-balancing policies are defined, followed by an overview of previous work in the field. In the last section, queueing models that provide the basis for the policies implemented in this thesis, are described.

2.1 Brief Overview of Taxonomy of Load Balancing Policies

In this section, the different categories of load-balancing policies are presented and can be found in [12, 20]. A detailed overview of the different taxonomies can be found in [12]. Figure 2.1 shows the organization of the different load-balancing schemes.

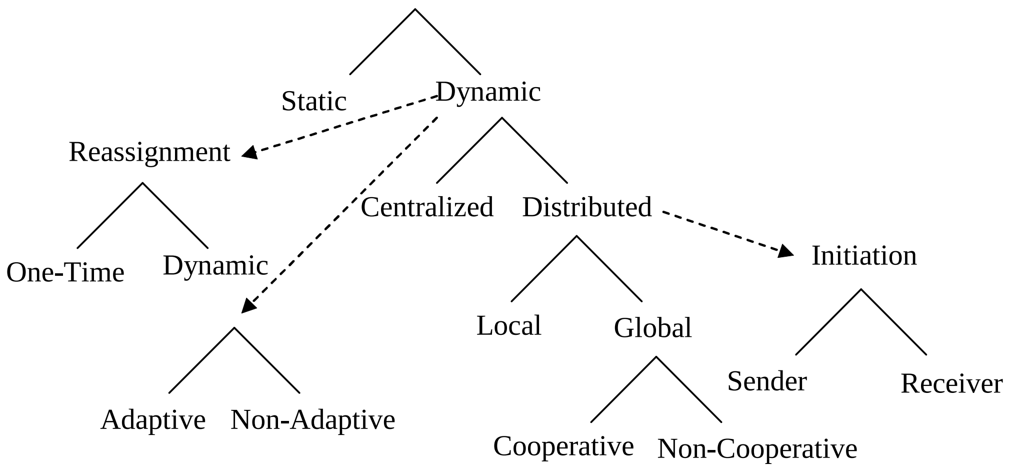


Figure 2.1: Load-balancing schemes.

2.1.1 Static Versus Dynamic

Static load distribution, also known as deterministic scheduling, assigns a given job to a fixed processor or node. Every time the system is restarted, the same binding task-processor (allocation of a task to the same processor) is used without considering changes that may occur during the system's lifetime. Moreover, static load distribution may also characterize the strategy used at runtime, in the sense that it may not result in the same task-processor assignment, but assigns the newly arrived jobs in a sequential or fixed fashion. For example, using a simple static strategy, jobs can be assigned to nodes in a round-robin fashion so that each processor executes approximately the same number of tasks.

Dynamic load-balancing takes into account that the system parameters may not be known beforehand and therefore using a fixed or static scheme will eventually produce poor results. A dynamic strategy is usually executed several times and may reassign a previously scheduled job to a new node based on the current dynamics of the system environment.

2.1.2 Distributed Versus Centralized

This division usually falls under the dynamic load-balancing scheme where a natural question arises about where the decision is made. Centralized policies store global information at a central location and use this information to make scheduling decisions using the computing and storage resources of one or more processors. This scheme is best suited for systems where an individual processor's state information can be easily collected by a central station at little cost, and new jobs arriving at this centralized location are then redirected to subsequent nodes. The main drawback of this scheme is that it has a single point of failure.

In distributed scheduling, the state information is distributed among the nodes that are responsible in managing their own resources or allocating tasks residing in their queues to other processors. In some cases, the scheme allows idle processors to assign tasks to themselves at runtime by accessing a shared global queue. Note that failures occurring at a particular node will remain localized and may not affect the global operation of the system.

Another scheme that fits between the two types above is the hierarchical one where selected nodes are responsible for providing task scheduling to a group of processors. The nodes are arranged in a tree and the selected nodes are roots of the subtree domains. An example of this scheme is described in Section 2.5.

2.1.3 Local Versus Global

Local and global load-balancing fall under the distributed scheme since a centralized scheme should always act globally. In a local load-balancing scheduling, each processor polls other processors in its neighborhood and uses this local information to decide upon a load transfer. This local neighborhood is usually denoted as the migration space. The primary objective is to minimize remote communication as well as efficiently balance the load on the processors. However, in a global balancing scheme, global information of all or part of the system is used to initiate the load-balancing. This scheme requires a considerable amount of information to be exchanged in the system which may affect its scalability.

2.1.4 Cooperative Versus Non-Cooperative

Within the realm of distributed dynamic global scheduling, two mechanisms can be distinguished involving the level of cooperation between the different parts of the system. In the non-cooperative or autonomous scheme, each node has autonomy over its own resource scheduling. That is, decisions are made independently of the rest of the system and therefore the node may migrate or allocate tasks based on local performance. On the other hand, in cooperative scheduling, processes work together toward a common system-wide global balance. Scheduling decisions are made after considering their effects on some global effective measures (for example, global completion time).

2.1.5 Adaptive Versus Non-Adaptive

Adaptive and non-adaptive schemes are part of the dynamic load-balancing policies. In an adaptive scheme, scheduled decisions take into consideration past and current system performance and are affected by previous decisions or changes in the environment. If one (or more parameters) does not correlate to the program performance, it is weighted less next time. In the non-adaptive scheme, parameters used in scheduling remain the same regardless of system's past behavior. An example would be a policy that always weighs its inputs the same regardless of the history of the system behavior.

Confusion may arise between in distinguishing dynamic scheduling and adaptive scheduling. Whereas a dynamic solution takes environmental inputs into account when making its decision, an adaptive solution (which is also dynamic) takes environmental stimuli into account to modify the scheduling policy itself [12]. An adaptive policy is proposed in Chapter 6.

2.1.6 One-Time Assignment Vs Dynamic Reassignment

In this classification, the entities to be scheduled are considered. The one-time assignment of a task may be dynamically done but once it is scheduled to a given processor, it can never be rescheduled to another one [23]. On the other hand, in the dynamic reassignment process, jobs can migrate from one node to another even after the initial placement is made. A negative aspect of this scheme is that tasks may endlessly circulate about the system without making much progress.

2.1.7 Sender/Receiver/Symmetrical Initiated

Techniques of scheduling tasks in distributed systems have been divided mainly into sender-initiated, receiver-initiated, and symmetrically-initiated. In sender-initiated algorithms, the overloaded nodes transfer one or more of their tasks to several under-loaded nodes. In receiver-initiated schemes, under-loaded nodes request tasks to be sent to them from nodes with higher loads. In the symmetric approach, both the under-loaded as well as the loaded nodes may initiate load transfers.

2.2 Previous Work

In this section, several load-balancing policies introduced in earlier works are described.

2.2.1 Shortest Expected Delay (SED) and Adaptive Separable policy (AS)

The shortest expected delay (SED) [7, 45] and adaptive separable policy (AS) [45] are based on the multiple queue multiple server model shown in Figure 2.2. Both policies can be either centralized where new tasks arrive to a central server and then assigned to subsequent nodes, or distributed where each available node can insert new jobs into the system. In any case, the algorithm is triggered whenever a new job arrives at node p. Subsequently, a cost function is evaluated for each node and the job is sent to the corresponding node that produces the minimum cost. the cost SED(i) is actually the expected time to complete the new job at node i and is given by

$$\mathrm{SED}(i) = \frac{n_i + 1}{\mu_i}, \tag{2.1}$$

where n_i and μ_i are respectively the load and service rate of node i. The information exchanged and the balancing process can be done either globally or is restricted to local domains.

The adaptive separable policy is an improvement over the SED policy in the sense that it estimates the completion time of new arrival at a node by adjusting the service rate based on its utilization or idle time fraction u_i. The new cost becomes

$$\mathrm{AS}(i) = \frac{n_i + 1}{\mu_i u_i}$$

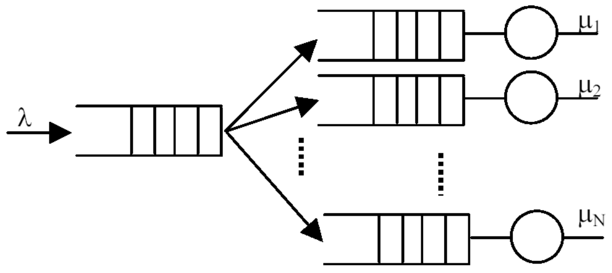


Figure 2.2: The multiple queue multiple server model. λ is the job arrival rate and μ_i is the service rate of node i

2.2.2 Never Queue Policy (NQ)

The never queue policy (NQ) ([39]) is inspired by the fact that in heterogeneous systems, fast servers may take over all the slow servers in executing most of the jobs and therefore result in idle nodes in the system. This case occurs mostly when applying the SED policy in a highly loaded environment and thus yields suboptimal results.

The NQ policy first assigns the newly arriving job to the idle node. If more than one idle node is available, the new job is sent to the fastest node that has the largest $(1/\mu_i)$ term. On the other hand, if all nodes are busy, the SED policy is used.

2.2.3 Maximum Throughput Policy (TP) and Greedy Throughput Policy (GT)

The aim of the maximum throughput policy developed by Chow and Kohler in 1979 [15] is to maximize the throughput of the system during the next job arrival. The throughput function TP is given by

$$\mathrm{TP}(n_1, n_2, \cdots, n_m) = \sum_{i=1}^{m} \lambda \left[\sum_{k=1}^{n_i - 1} \left(1 - \frac{n_i}{k}\right) \left(\frac{\mu_i}{\lambda + \mu_i}\right)^k - n_i \ln\left(\frac{\lambda}{\lambda + \mu_i}\right) \right], \quad (2.2)$$

where λ is the arrival rate and m is the number of nodes in the system. TP is a reward function calculated for each possible assignment for the new arriving job and the node that maximizes this function is chosen. This function is complex to evaluate and renders the load-balancing algorithm inefficiently slow.

Nelson and Towsley in 1985 derived another reward function that is implemented in the Greedy Throughput policy (GT). The GT reward function stated in [40] is easier to evaluate than TP and is given as,

$$\mathrm{GT}(i) = (\frac{\mu_i}{\mu_i + \lambda})^{n_i+1} \tag{2.3}$$

Both TP and GT policies depend on the inter-arrival rate λ which may not be available in a real system implementation.

2.2.4 The Gradient Model

In the gradient model policy [30], the underloaded nodes notify the other nodes about their state, and overloaded nodes respond by transmitting jobs to the nearest lightly loaded node. Therefore, loads migrate in the system in the direction of the underloaded nodes guided by the proximity gradient. A global balance state is achieved computationally by successive localized balances.

At every step of the algorithm, each node compares its load to a Low-Water Mark (LWM) and a High-Water Mark (HWM) thresholds. The node is set to the underloaded state if it has a load less than LWM and to the overloaded state if it has a load greater than HWM. Underloaded nodes set their proximity to zero and all other nodes p set their proximity according to

$$\mathrm{proximity}(p) = \min(\mathrm{proximity}(n_i)) + 1 \tag{2.4}$$

where n_i denote the neighboring nodes of node p. The node's proximity is defined as the shortest distance from itself to the nearest lightly loaded node in the system.

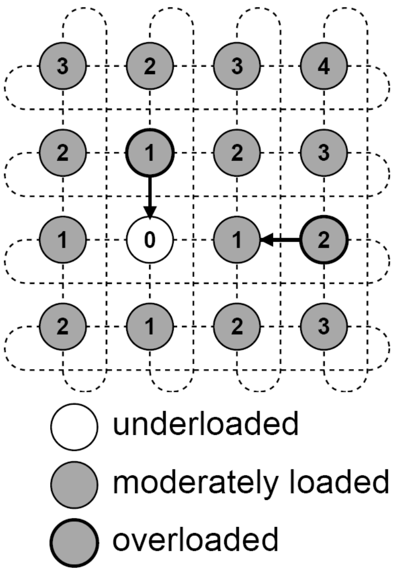


Figure 2.3: Gradient model example.

Subsequently, all overloaded nodes send a fraction δ of their loads in the direction of the lowest proximity. The algorithm is illustrated in Figure 2.3.

Note that no measure of the degree of imbalance is found using this algorithm, but only that one exists. When an imbalance occurs, the number of excess tasks can only be known to be greater than HWM-LWM. Hence, the HWM, LWM, and the fraction δ parameters have a critical impact on the stability and performance of the algorithm and should therefore be wisely chosen.

The gradient model policy cannot be used in distributed systems since the nodes are not connected in a certain topology such as a mesh or hypercube. This fact renders the proximity concept useless. Moreover, the proximity algorithm is a cascading function and therefore requires a considerable amount of time to be evaluated in large-scale networks where delays are prominent.

However, a modification to the algorithm may be suitable for P2P networks such as Freenet where nodes are only aware of their immediate neighbors. Consequently, the proximity concept becomes valid and the algorithm may become useful.

2.2.5 Sender Initiated Diffusion (SID) and Receiver Initiated Diffusion (RID)

Sender initiated diffusion ([46][36]) and receiver initiated diffusion ([46][41]) are local strategies based on the near-neighboring diffusion concept. Each node exchanges information within its own domain composed of a node and its neighboring nodes. Global balancing is achieved by the fact that the domains are overlapping.

For the SID policy, the balancing process is triggered whenever a node p receives from a neighboring node i a load update l_i less than a preset threshold L_{low} ($l_i < L_{low}$). After that, the node p proceeds by calculating the domain load average $\overline{L}_p$

$$\overline{L}_p = \frac{1}{K+1}\left(l_p + \sum_{k=1}^{K} l_k\right) \tag{2.5}$$

where K is the number of neighboring nodes. The load balancing algorithm continues if the local excess load ($l_p - \overline{L}_p$) is greater than a preset threshold $L_{threshold}$. Load δ_k is then transferred from node p to each neighbor in proportion to its deviation from the domain calculated using

$$\begin{aligned} h_k &= \begin{cases} \overline{L}_p - l_k & \text{if } l_k < \overline{L}_p, \\ 0 & \text{otherwise.} \end{cases} \\ \delta_k &= (l_p - \overline{L}_p)\frac{h_k}{\Sigma_{k=1}^{K} h_k} \end{aligned} \tag{2.6}$$

The RID strategy can be thought of as the converse of the SID strategy in that it is a receiver-initiated approach as opposed to the sender-initiated approach [46]. However, to avoid instability due to delays and aging in the load exchange information, the overloaded nodes transmit tasks up to the half of their current load. SID and RID are illustrated in Figure 2.4.

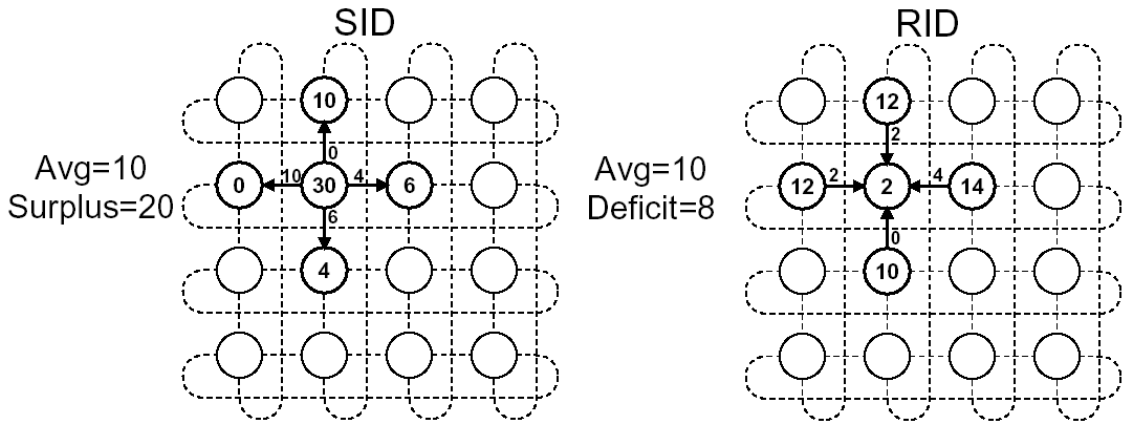


Figure 2.4: Sender Initiated Diffusion (SID) and Receiver Initiated Diffusion (RID examples

This scheme is distributed, asynchronous, and topology independent as opposed to the GM policy that is best suited for nodes arranged in a hypercube or mesh fashion. However, the same problem arises in defining the local domain where the balancing process should take place. Moreover, as indicated earlier, the domains should overlap to an extent that is sufficient to achieve global balancing.

The algorithms implemented in this thesis are based on the SID scheme without restricting the balancing process to a local domain, but rather expanding it to the global system.

2.2.6 Hierarchical Balancing Method (HBM)

The Hierarchical Balancing Method (HBM) strategy [46] arranges the nodes in a hierarchy, thereby creating balancing domains at each level. For a binary tree organization, all nodes are included at the leaf level (level 0). Half the nodes at level 0 become subtree roots at level 1. Subsequently, half the nodes again become subtree roots at the next level and so forth until one node becomes the root of the whole tree.

Global balancing is achieved by ascending the tree and balancing the load between adjacent domains at each level in the hierarchy. If at any level, the imbalance between the left and right subtrees exceeds a certain threshold, each node in the overloaded subtree sends a portion of its load to the corresponding node in the underloaded subtree.

The advantage of the HBM scheme is that it minimizes the communication overhead and therefore can be scaled to large systems. Moreover, the policy matches hypercube topologies well. In fact, the dimensional exchange approach [17] designed for hypercube systems is similar to the HBM method in the sense that it proceeds by load-balancing per domain basis. Here, each domain is defined as one dimension in the hypercube. The hierarchical organization of an eight-processor hypercube is shown in Figure 2.5.

This scheme is clearly not suitable for systems with large network delays for the following reasons. As the balancing process proceeds on to the next level in the tree, critical changes occurring at lower levels may not propagate quickly due to delays. Therefore, corrections may not reach higher domains in time and may thereby result in an imbalance at the global level. Moreover, although the scheme is decentralized, a failure at root nodes especially at high levels in the tree, renders a global balance state unattainable. Consequently, this scheme is not suitable for Internet-scale distributed systems since nodes may become unreachable at any time, and will therefore affect the balance state of the system if such nodes happen to be roots for subtree domains.

2.2.7 Simulations and Modifications

SED, NQ, TP GT and AS policies were compared by Banawan and Zeidat in [8]. Several simulations were performed on different types of systems. These systems vary

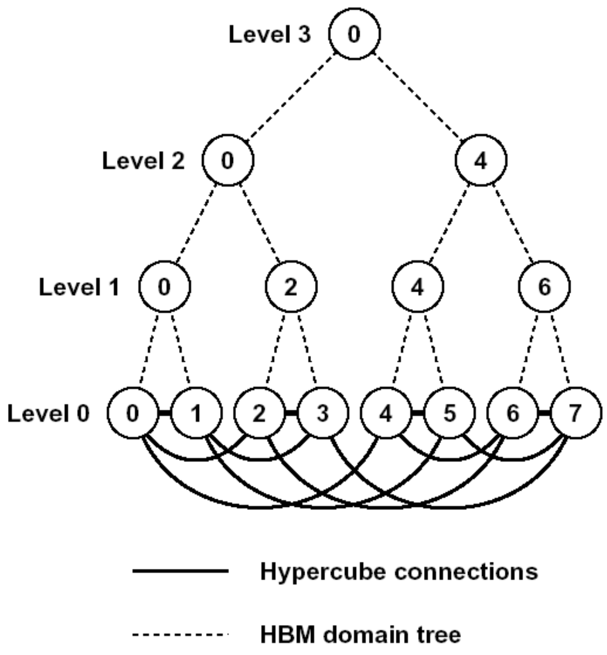


Figure 2.5: Hierarchical organization of an eight-processor system with hypercube interconnections. The processor ID's at intermediate nodes in the tree represent those processors delegated to manage the balancing of corresponding lower-level domains.[46]

by their node service rates μ_i, system utilization, and network delays. The results indicate that in most cases, the NQ policy performed best.

In a recent paper, Kabalan et al. [28] introduced modifications to the SED, NQ and GT policies in order to account for the delay incurred in transferring jobs between nodes. The term $t_{comp}(i,j)$ representing the communication delay between node i and node j is added to the costs calculated in the SED and NQ policy. Furthermore, a q_{max} threshold on the queue sizes of the nodes is considered. No new jobs are transferred to nodes having queue sizes more than q_{max} even though they may have the least cost associated with them. In the latter case, the “second best” node is chosen.

For the GT policy, an empirical method for calculating the λ rate was proposed where it was assumed that the delay in transferring a task is less than the inter-arrival time of new jobs. Simulations were conducted over eight heterogeneous nodes positioned according to a fixed topology. The results show that the NQ policy outperformed the other policies under most operating system conditions.

On the other hand, Willebeek and Reeves in [46] simulated the GM, RID, SID, HBM and the Dimension Exchange Method (DEM) policies on a 32-processor 5-dimensional hypercube Intel iPSC/2 machine. Their results show that low granularity tasks gave poor results due to lower ability to optimally transfer loads. Also high tasks granularity also gave poor results due to the increased overhead of moving tasks. Nevertheless, The DEM and HBM policies gave the best results as expected. However, the authors concluded by recommending the RID scheme that surprisingly gave good results for a broader range of systems (non-hypercube).

2.3 Load-Balancing Models

In this section we describe two queueing models for local, sender-initiated, load-balancing algorithms that were developed at the University of New Mexico and the University of Tennessee. These models were initially tested in simulations, and the system developed in this thesis has been used to validate both models in a real environment under different policies.

Both models focus upon the effects of delays in the exchange of information among the computational elements (CEs), and the constraints these effects impose on the design of a load-balancing strategy.

2.3.1 Deterministic Time Delay Queueing Model for Load-Balancing

The deterministic time model is a continuous-time described in terms of of a nonlinear delay-differential system [5, 10]. It also considers deterministic communication and transfer delays.

The authors consider a computing network consisting of n nodes all of which can communicate with each other. Initially, the nodes are assigned an equal number of tasks. However, when a node executes a particular task it can generate more tasks so that the overall load distribution becomes non-uniform. To balance the loads, each computer in the network sends its queue size $q_j(t)$ at time t to all other computers in the network. A node i receives this information from node j delayed by a finite amount of time τ_{ij}, that is, it receives $q_j(t - \tau_{ij})$. Each node i then uses this information to compute its local estimate of the average number of tasks per node in the network using the simple estimator $\left(\sum_{j=1}^{n} q_j(t - \tau_{ij})\right)/n$ ($\tau_{ii} = 0$), which is based on the most recent observations. Node i then compares its queue size $q_i(t)$ with its estimate of the network average as $q_i(t) - \left(\sum_{j=1}^{n} q_j(t - \tau_{ij})\right)/n$ and, if this is greater than zero, the node sends some of its tasks to the other nodes while if it is less than zero, no tasks are sent. Furthermore, the tasks sent by node i are received by node j with a delay h_{ij}. The authors present a mathematical model of a given computing node for load-balancing, which is given by:

$$\begin{aligned}
\frac{dx_i(t)}{dt} &= \lambda_i - \mu_i + u_i(t) - \sum_{j=1}^{n} p_{ij} \frac{t_{p_i}}{t_{p_j}} u_j(t - h_{ij}) \\
y_i(t) &= x_i(t) - \frac{\sum_{j=1}^{n} x_j(t - \tau_{ij})}{n} \\
u_i(t) &= -K_i \text{sat}\,(y_i(t)) \\
p_{ij} &\geq 0, p_{jj} = 0, \sum_{i=1}^{n} p_{ij} = 1
\end{aligned} \tag{2.7}$$

where

$$\begin{aligned}
\text{sat}\,(y) &= y \text{ if } y \geq 0 \\
&= 0 \text{ if } y < 0.
\end{aligned}$$

In this model:

- $x_i(t)$ is the expected waiting time experienced by a task inserted into the queue of the i^{th} node and $u_i(t)$ is the rate of removal (transfer) of the tasks as determined by the balancing algorithm.
- λ_i is the rate of increase in x_i
- μ_i is the service rate at the i^{th} node
- p_{ij} decides the fraction to be sent from node j to node i

local information of the waiting times $x_i(t), i = 1, .., n$ are used to set the values of the p_{ij} such that node j can send tasks to node i in proportion to the amounts by which node i is below the local average as seen by node j. Several methods can be used to choose the p_{ij}'s according to predefined policies. These policies will be discussed in the next chapter.

2.3.2 Stochastic Time Delay Queueing Model for Load Balancing

In this section, a stochastic time delay queueing model in differential form is described [24, 20, 19]. The motivation behind this model is the stochastic nature of the distributed computing problem that include: 1) Randomness and possible burst-like nature of the arrival of new job requests at each node from external sources (i.e., from users); 2) Randomness of the load-transfer process itself, since the communication delays in large networks are random; and 3) Randomness in the task completion process at each node. Based on these facts, the following dynamics of the ith queue in differential form is given by

$$Q_i(t+\Delta t) = Q_i(t) - C_i(t, t+\Delta t) - \sum_{j \neq i} L_{ji}(t) + \sum_{j \neq i} L_{ij}(t - \tau_{ij}(t)) + J_i(t, t+\Delta t), \quad (2.8)$$

where

- $C_i(t, t + \Delta t)$ is a Poisson process with rate μ_i describing the random number of tasks completed in the interval $[t, t + \Delta t]$
- $J_i(t, t+\Delta t)$ is the random number of new (from external sources) tasks arriving in the same interval, as discussed above
- $\tau_{ij}(t)$ is the delay in transferring the load arriving to node i in the interval $[t, t + \Delta t]$ from node j, and finally
- $L_{ij}(t)$ is the load transferred from node j to node i at the time t.

For any $k \neq \ell$, the random load $L_{k\ell}$ diverted from node ℓ to node k is governed by the load-balancing policy at hand. In general,

$$L_{kl}(t) = K_k p_{kl} \cdot \left(Q_l(t) - n^{-1} \sum_{j=1}^{n} Q_j(t - \eta_{lj}(t)) \right) \cdot u\left(Q_l(t) - n^{-1} \sum_{j=1}^{n} Q_j(t - \eta_{lj}(t)) \right),$$

where $u(\cdot)$ is the unit step function, $\eta_{lj}(t)$ is the state exchange delay between the jth and lth nodes at time t and K_k is the gain parameter at the kth (load distributing) node. The fractions p_{ij} will be discussed in the next chapter as part of the load-balancing strategy.

2.4 Summary

In this chapter, a number of load-balancing policies and their taxonomies were described. Moreover, the queueing models describing the behavior of the system were presented. In the next chapter, the test-bed software that implements several load-balancing policies based on these models is introduced, followed by a description of the different strategies that were actually adopted and experimented.

Chapter 3

Implementation Architecture

A distributed system has been developed to validate the deterministic model and the stochastic model described in Sections 2.3.1 and 2.3.2 and to assess the performance of different load-balancing policies in a real environment. The system consists of duplicates of the same software running on each node. The load-balancing decision consisting of when to balance and how many tasks to transmit, is done locally at each node. The decision is therefore distributed as opposed to be centralized, case for which a master node is responsible for making the decision. The load-balancing process running on each node bases its decision on local information and on shared data which are exchanged between the nodes. The initial configuration of each node is set through three configuration files, which will be discussed in Section 3.7.

In this chapter, the internal architecture of the load-balancing distributed system is described.

3.1 Platforms

The load-balancing software was built in ANSI C over UNIX-based systems, namely, Sun Solaris and Linux. Sun machines were used to run experiments over the LAN network in the ECE department whereas the Planet-Lab system was used to run experiments over the Internet. The Planet-Lab [2] operating system is based on the Linux RedHat operating system. On the other hand, in order to run experiments over the wireless test-bed, the code was imported to the "Cygwin" environment that runs over Microsoft Windows. Cygwin [1] is a Linux-like environment for Windows that acts as a Linux emulation layer, providing substantial Linux API functionality.

The system has a multi-threaded architecture where the POSIX-threads programming standard was used. BSD socket mechanism was used for the network programming aspect of the system. References [42],[16] and [43] were extensively used when the load-balancing system was implemented.

3.2 Macro-Architecture

The general architecture of the system consists of three layers as shown in Figure 3.1. Each layer is implemented as a module in order to facilitate its own modification or replacement without affecting the other layers. More importantly, this architecture allows the testing and implementation of different load-balancing policies by simply changing few lines of code and without interfering with the rest of the system layers. The modules communicate with one another through well-defined interfaces. In what follows, a detailed description of the system architecture is provided that is summarized in Figure 3.7.

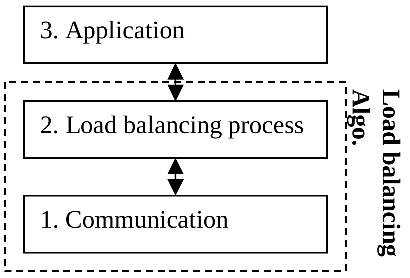


Figure 3.1: Load-balancing system architecture.

3.3 Variables and Data Structures

Two main data structures were used in the program. The first one is a simple linked-list that contains state information about the rest of the nodes and also used as a communication tool between the load-balancing module and the task transmission module. This list, illustrated in Figure 3.2, is created upon executing the program and no subsequent alteration is made except for the information stored inside of it. The "state information" is mostly kept up-to-date by the "state reception" module, and it contains information regarding the node *current* queue size, computational power, etc. These parameters are stored in the *info* structure that is shown below.

```
    //structure stored at node i and contains information about node j
struct info{
 long q_size;                        //Latest queue size of node j received
 struct timespec timestamp;         //time-stamp set by node j when
                                    //the latest state was transmitted
 struct timespec local_timestamp; //node i time-stamp when the state
                                    //information was received
 unsigned long rate;                //(bytes/s) bandwidth detected between
                                    //node i and node j
 unsigned long symm_rate;          //(bytes/s) bandwidth detected between node j and node i
 struct timespec C;                //average task execution time of node j (nano-second)
 long data_ID;                      //ID of the latest frame of tasks transmitted to node j
 long rec_data_ID;                 //ID of the latest frame of tasks received from node j
 long exp_data_ID;                 //expected frame of tasks that should be received
```

```
                                //from node j
}
```

The way these parameters are calculated and used are policy-dependent and therefore will be discussed in subsequent sections.

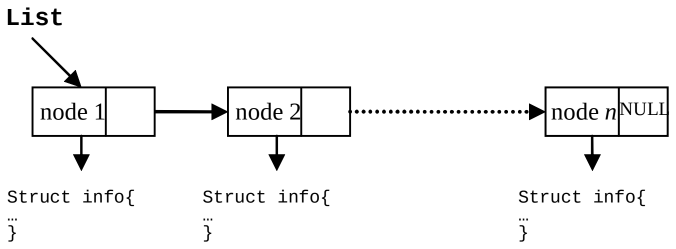


Figure 3.2: List data structure containing others node state information

The second data structure used in the program is the task queue, which has a linked-queue structure as illustrated in Figure 3.3. Newly arriving tasks from either an external source or from within the network (sent by other nodes) are added to the rear of the queue, whereas the application at hand pops one task at a time from the front of the queue and executes it. Moreover, the load-balancing layer may decide at any time to transfer several tasks to other nodes and therefore extract from the front of the queue the desired number of tasks that it wishes to transmit. In all cases, for any operation applied on the task queue, the variable *Current_Queue_Size* that reflects the number of tasks present in the queue, is *atomically* updated accordingly.

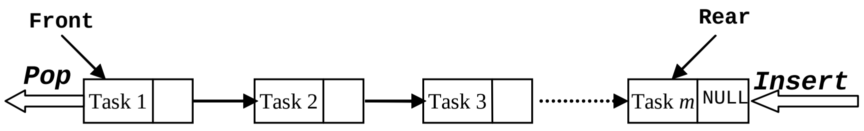


Figure 3.3: Queue structure holding tasks information.

3.4 Communication Layer

The communication layer is divided into four separate threads: the “state transmission,” the “state reception,” the “tasks reception,” and the “tasks transmission” threads. The “state transmission” thread is responsible for transmitting the state of the local node to all other nodes that are part of the system. The state information includes the information listed in the previous section which incorporates the current queue size, the node computational power, and other local information that may be relevant to the load-balancing policy in use. The sizes of the state frame ranges between 20 and 34 bytes depending on the policy at hand. The transmission of the node state is performed every defined amount of time as specified in the node initialization file. As for the transport protocol, one has the option of either TCP or UDP by setting the appropriate parameter in the initialization file. It is always recommended to use UDP since it involves less overhead in the transmission. In case the state frame was dropped by the network, a retransmission will occur in the next scheduled state exchange. The “state transmission” can also be triggered by the load-balancing thread if the policy in use decides so.

The “state reception” thread is the complement of the “state transmission” thread and has the architecture of a concurrent single threaded server. The single threaded architecture was used because our aim is to provide maximum performance to the application layer that is running in the same process, which will be slowed down if more threads were created. The “state reception” listens to a well-defined TCP or UDP port. Upon reception of any state information, the thread updates the corresponding node information, available in the local node-list, whenever the time-stamp of the received frame is greater than the time-stamp of the stored information. This is done to ensure that two state frames will not overwrite each other, if they were received in reverse order. This scenario may happen frequently in packet switching networks.

The “tasks transmission” module responsible for transmitting jobs to other nodes, runs in the same thread as the load-balancing module that will be described later. The main reason behind this design is that another instance or cycle of the load-balancing policy cannot be initiated unless all prior data transmissions have been completed. The “tasks transmission” module also has concurrent single threaded client architecture. Only TCP can be used as a transport protocol since reliable transmission is needed so that no tasks are lost in the network. Upon transmission completion, the “task transmission” module informs the “load-balancing” module about its final status i.e., whether all, part, or none of the transmissions have been successful.

The “tasks reception” thread is the complement of the “tasks transmission” thread. It listens to a well-defined TCP port and accepts tasks sent from other nodes. Upon transfer completion, the “tasks reception” thread hands over the data received to the “application input” thread that is described later on. It also has concurrent single threaded server architecture for the same reasons stated above.

3.5 Load-Balancing Layer

The second layer of the system consists of a single thread called the “load-balancing” thread, which is the core of the program. This layer is easily modifiable to include different policies. Nevertheless, all policies follow the same general steps, identified by a cycle and described as follows. The load-balancing “process” is initiated at a predefined amount of time (read from a file) or at a calculated amount of time depending on the policy at hand. Consequently, the process determines the portion of the tasks to be sent to every node in the system if applicable. This decision is policy dependent and is based on the current state of the node and on the states of the other available nodes. Furthermore, some policies may also rely on the detected network delays, to calculate the number of tasks to transmit (see Chapter 5). Subsequently, the thread packs the tasks into a network frame; it has access to the tasks queue where it can extract the desired number of tasks without deleting them but setting their status to inactive. This is done in order to prevent the application from executing the tasks in the transition period. The purpose of this procedure is that after the "task transmission" module has completed its job, the tasks are either resetted to the active mode or deleted from the queue, depending on the status of the transmission (successful or not). As denoted earlier, another cycle of the load-balancing procedure cannot be initiated until all prior transmissions have completed.

The load-balancing policy is lower bounded by $\Omega(n)$ (Ω is a lower order execution time function) runtime execution where n is the number of nodes in the system. This is due to the fact that a traversal of the entire node-list is required in order to calculate the portion of the excess tasks to be sent to every node. In fact, the policies implemented in this thesis will be shown to have a runtime upper bounded by $O(n)$.

3.6 Application Layer

The application layer is divided into two threads: the "application input" and the "application execution" threads. The "application input" creates a number of tasks defined in the initialization file upon program startup and inserts them in the task queue. Moreover, this thread is responsible for adding new tasks to the queue either through an external source or from other nodes in the system. In the latter case, the "application input" gets the network frame from the "tasks reception" thread, unpacks it, and then adds the resulting tasks to the queue. On the other hand, the "application execution" thread is responsible for the tasks execution. It simply pops an active task from the queue, executes it, and then updates the *Current_Queue_Size* variable.

The above description applies to any generic application that can be divided into independent tasks. In our case, we used matrix multiplication as the basis for our experiments, where one task is defined as the multiplication of a row by a static matrix duplicated on all nodes. Therefore, the task queue contains rows having the same size, which can be set by a parameter in the initialization file. In order to emulate a real life application where the execution time of a task may vary, the size of each element (in bytes) of a single row is generated randomly from a specified range also set in the initialization file. This way, the multiplication of two elements or two numbers of different sizes may take different amounts of time, which leads, in turn, to variation in the execution time of the tasks.

To accomplish this objective (variability in the task execution time), arithmetic operations at arbitrary precision were performed. Instead of using the standard data types "int," "long int," "float" or "double" which have fixed precisions for storing numbers, a customized data type was created to allow arbitrary precision. The numbers are therefore stored in an *unsigned_char* array (each element of the array occupies 1 byte) having length N and interpreted in the radix (base) 256. N therefore denotes the precision of the number. The basic arithmetic operations (addition and multiplication) for arbitrary precision were acquired from [35]. They achieved $O(N \times \log N \times \log \log N)$ operations in multiplying two strings of length N. The trick was to recognize that multiplication is essentially a *convolution* of the digits of the multiplicand and multiplier, followed by some kind of carry operation [35]. The convolution is done by using the fast Fourrier transform (FFT). Essentially, what interests us is that modifying the precision N of the numbers we obtain different runtime in their multiplication or addition. We thereby constructed the row-matrix multiplication based on these operations and generated randomly the precision of each element in each row. An example of a row is shown in Figure 3.4. In this manner, different tasks will exhibit different completion times.

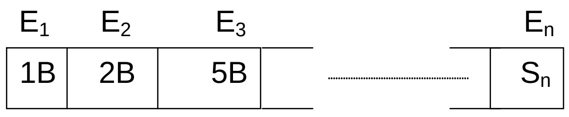


Figure 3.4: Example of a row of size n where the maximum precision was set to 5 Bytes.

3.7 Configuration and Log Files

Finally, the program has three initialization files; the parameter initialization file "init.ini" that contains policy and application related parameters, the balancing instance file"balance.ini" and the node file "node.ini" that contains the address of the nodes (either IP address or host-name) that are part of the system. An example of an initialization file is shown below where each parameter is explained by a commented description (% denotes a comment) that precedes it.

```
% the following parameter is the synchronization or the communication interval
% between 2 consecutive state frame broadcast
% declared in s and ns (seconds and nano seconds)

SYNC 1s 50000000ns

% the following parameter is the gain K used by the policy
% when deciding the total number of tasks to transmit
% usually nb_tasks_to_transmit=GAIN*excess_load
% must be less than 1

GAIN 0.7

% the following parameter defines the type of protocol used to send
% the state frame to other nodes.  It is either TCP or UDP
% (data (or tasks) exchange is done over the TCP protocol only)

SYNCPROTOCOL UDP

% the following parameter defines the initial number of tasks in the task-queue
% used by the application layer

INITNBTASKS 250

% the following parameter is the external task insertion rate
% used by the application layer to generate rows at that rate
% declared in s and ns (seconds and nano seconds)
% if it is set to zero, this feature is disabled

INPINTERVAL 0s 0ns

% the following paravmeter defines the number of elements in a single row
% used by the application layer

ROWSIZE 100

% the following parameter defines the maximum number of
% bytes (or precision) of a row element - used by the application layer
% this parameter has a direct influence on the execution time of a single task

MAXBYTES 15

% this parameter controls the occurrence of the load-balancing process.
% if it is set to YES, the program will delay the balancing instance
% if it knows that any of the other nodes is transferring loads to it.
% i.e if rec_data_ID ≠ exp_data_ID the load-balancing process is delayed.
% parameter value is either YES or NO

DELAYBALANCE NO
```

The“balance.ini” file contains a list of intervals that are used to initiate the load-balancing process. Each entry is defined in seconds and nano-seconds.

Furthermore, every operation performed by each node is logged to a file that is later used for statistical analysis and to generate plots. Eight types of logs corresponding to eight different events are described as follow:

1. This type is generated by the application layer and corresponds to a task completion event. The log includes: Time when the event happened, the corresponding task ID and the execution time (nano-second resolution).
2. This type is triggered by a change in the task queue size. The cause may be either the execution of a task, the transmission of one or several tasks, the reception of one or several tasks from the system, or an external source. The log includes: The time when the event took place and the resulting queue size.
3. This type corresponds to the initiation of the load-balancing process. The time when the event has occurred is recorded.
4. This type logs the event for tasks transmission attempt. The log includes: The time when the transmission began, the destination node IP address, the number of tasks to transmit, and the total tasks size (in bytes).
5. This type corresponds to the completion of the tasks transmission. It has the same fields as the previous type with the addition of the end transmission time.
6. This type corresponds to the tasks reception event. The log includes: The time when the tasks frame was received, the source node IP address, the number of tasks received and the corresponding size (in bytes).
7. This type corresponds to a state transmission event. The state of the node is recorded in addition to the corresponding IP address of the destination node and the time when the transmission has occurred.
8. This type corresponds to a state reception event. The state of the source node is recorded in addition to the time when the state was received.

3.8 Policies Implemented

The policies implemented in this system follow the same general guidelines but differ mainly in the scheduling of the load-balancing process and the allocation of the fractions p_{ij}. Recall from the previous chapter that p_{ij} is the fraction of the excess tasks as decided by node j that will be transmitted to node i.

The first scheme is the one-shot load-balancing [20, 19] where the nodes attempt to exchange tasks among themselves only once. The scheduling of this single-balancing instance is usually done early after the launch of the system but not before the state information of each node has widely propagated. This is done to ensure that each node is aware of the state of the other nodes when deciding on its load distribution strategy. This scheme is mostly suitable in systems where external arriving tasks are not prominent and the servicing rate or the computational power of each node is, more or less, stable. In any case, this scheme can be extended to the latter cases where a new balancing instance can be scheduled according to the occurrence of a special event such as the arrival of a new external task as proposed in [20]. Experimental work has been done to find the optimal balancing instance for the single-shot load-balancing strategies (Section 5.2) [23].

The second scheme allocates regularly a balancing instance when the load distribution process is triggered and tasks exchange between nodes take place [22]. In our case, the balancing instants are read from the "balance.ini" initialization file that was introduced in the previous section. The time intervals between two consecutive balancing instances may be constant or varying. Since the load distribution policy is distributed, each node can choose its balancing instances differently from the others as defined in each node's "balance.ini" file.

In both scheduling schemes, whenever the load-balancing process is triggered at node j in a system of n nodes, the following steps occur.

1. Node j calculates the total number of tasks available in the system from the information available in its node-list.

   $$Queue\_total = \sum_{k=0}^{n} Queue(k)$$

2. `Set` $Queue\_average = Queue\_total/n$

   `If` $Queue(j) < Queue\_average$

   `Then:` exit the process and wait until the next balancing instance

   `Else:` continue to the following step.

3. Calculate the total excess load to be transmitted by node j.

   $$Excess\_tasks = (Queue(j) - Queue\_average) * K,$$

   where K is a gain parameter.

4. Calculate the fractions p_{ij} of the excess tasks that node j will transmit to node i. Three different methods can be used.

   a. constant p_{ij}

   $$p_{ij} = 1/(n-1) \tag{3.1}$$

   b.

   $$p_{ij} = \begin{cases} \frac{Queue\_average - Queue(i)}{\sum_{k=1,k\neq j}^{n}(Queue\_average - Queue(k))} & \text{if } Queue(i) < Queue\_average, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

   c.

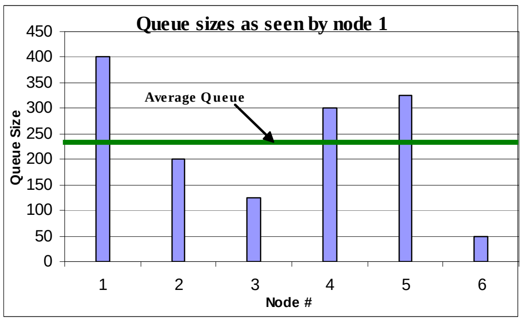
   $$p_{ij} = \frac{1}{n-2}\left(1 - \frac{Queue(i)}{\sum_{k=1,k\neq j}^{n} Queue(k)}\right) \tag{3.3}$$

   if no information about one or any of the nodes is available to node j , p_{ij} is set to $1/(n-1)$ for all i

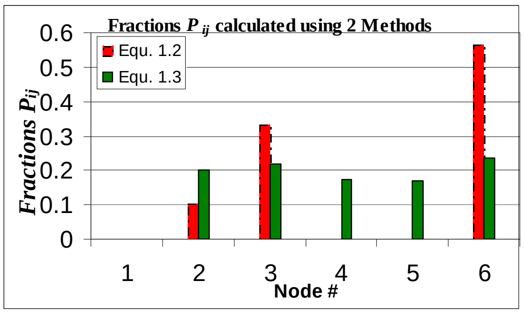
5. Transmit $p_{ij} * Queue\_excess$ tasks to node i.

One may think that setting the gain parameter $K = 1$ will achieve the best performance. But in systems with large delays where nodes may rely on outdated information in calculating the load distribution, $K = 1$ will actually give poor results. This phenomenon was first observed by the load balancing group at the University of New Mexico and The University of Tennessee in their simulation and analytical work. The experiments described in chapter 5 have been performed in order to optimize over the gain values K.

Equations (3.1) and (3.2) used for setting the fractions p_{ij} were introduced in [13] and [10]. These equations were primarily used in simulations and experiments to validate the deterministic model of Section 2.3.1. On the other hand, Equation (3.3) was used in simulations and experiments to validate the stochastic model of section 2.3.2 [24, 18]. Note that both methods allocate tasks to nodes inversely proportional to their queue sizes. The two methods are illustrated in Figure 3.5. A block diagram of the policies is shown in Figure 3.6.



(a) Queue sizes of the nodes as stored in the node-list of node 1 at the time when the load-balancing policy was initiated at node 1



(b) Fractions p_{ij} as calculated by node 1 using the two different methods of Equations 3.2 and 3.3.

Figure 3.5: Example of load distribution policies performed by node 1.

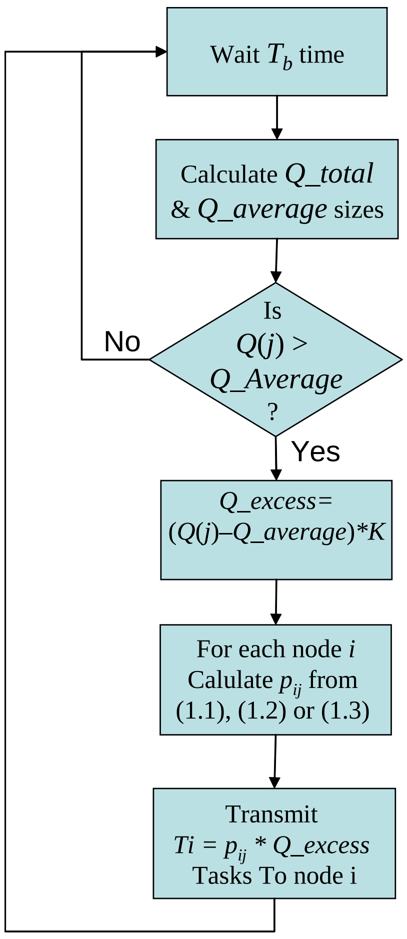
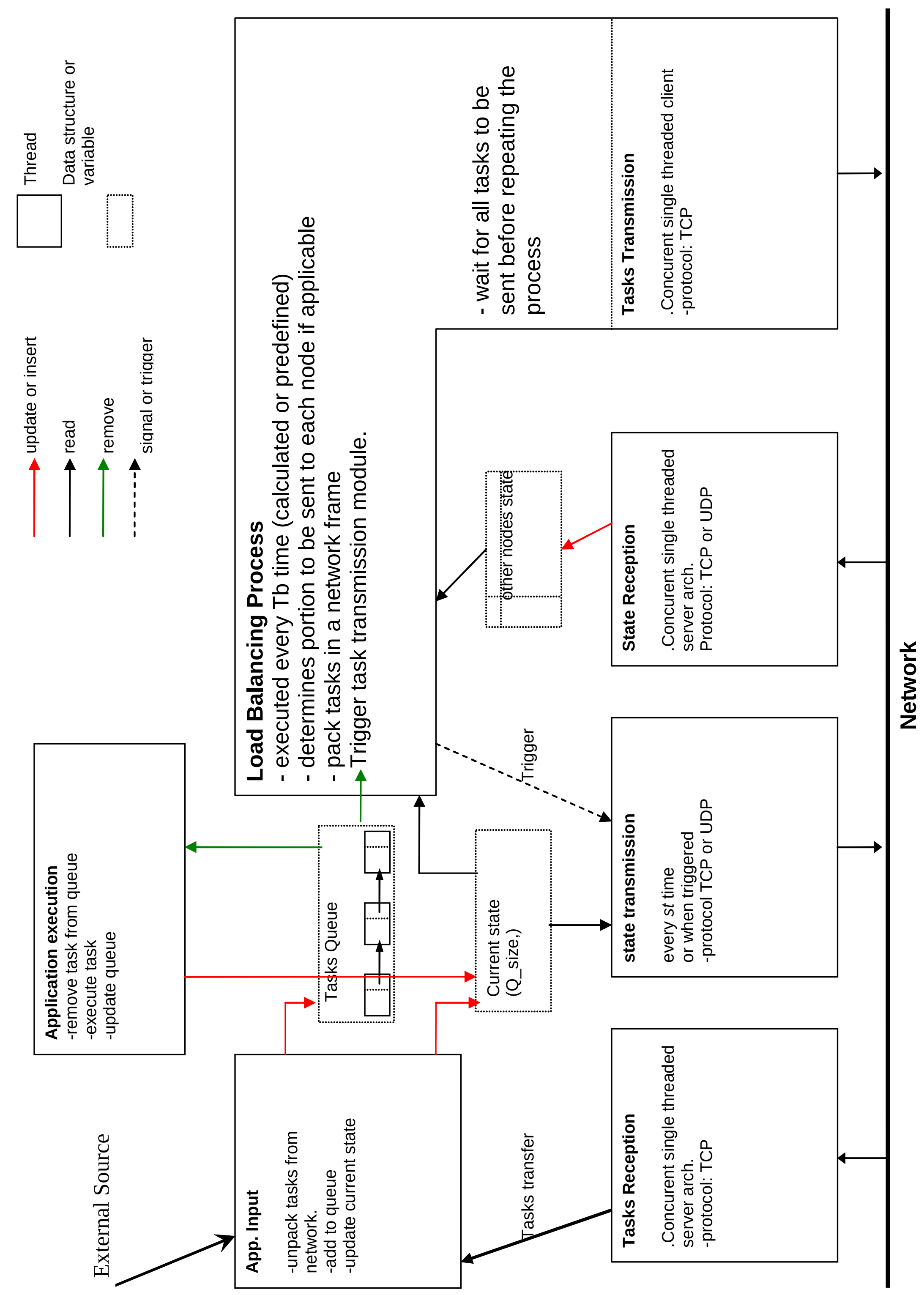

Wait T_b time
Calculate Q_total & $Q\_average$ sizes
Is $Q(j) >$ $Q\_Average$?
No
Yes
$Q\_excess = (Q(j) - Q\_average)*K$
For each node i Calulate p_{ij} from (1.1), (1.2) or (1.3)
Transmit $Ti = p_{ij} * Q\_excess$ Tasks To node i

Figure 3.6: Summary of the steps for the load-balancing policy performed at node j.

We can deduce from steps 1-5 that the algorithm scales linearly with the number of new nodes added to the system. The runtime is therefore $O(n)$ due to the fact that a full traversal of the node-list is needed in steps 1 and 4. A more advanced algorithm is developed in chapter 6 where network information is used to determine the portions p_{ij}.

3.9 Summary

A flexible software architecture of a distributed system for different load-balancing policies is introduced then followed by a description of the strategies that were actually employed. The functionality of this system has been implemented and tested in several environments and under different conditions. These environments include: wireless networks with infrastructure, wireless Ad-Hoc networks, Local Area Networks (LAN) and the Internet. The purpose of this variety of environments is to test certain load-balancing policies under different types of delays i.e., different pdf's (probability density function) for the delay and different transmission rates. Moreover, the system was also tested over Planet-Lab [2], a planetary-scale network involving more than 350 nodes positioned around the globe and connected via the Internet. The advantage of testing over such a platform is to determine the scalability of our system and its reaction to variable and broader range of delays. The testing results are detailed in Chapter 5 whereas Chapter 4 provides experimental investigations of delay in different types of networks in order to better understand how it may affect the load distribution strategies.



Thread
Data structure or variable
update or insert
read
remove
signal or trigger
Load Balancing Process
- executed every Tb time (calculated or predefined)
- determines portion to be sent to each node if applicable
- pack tasks in a network frame
- Trigger task transmission module.
- wait for all tasks to be sent before repeating the process
Tasks Transmission
.Concurent single threaded client
-protocol: TCP
other nodes state
State Reception
.Concurent single threaded server arch.
Protocol: TCP or UDP
Trigger
state transmission
every st time or when triggered
-protocol TCP or UDP
Application execution
-remove task from queue
-execute task
-update queue
Tasks Queue
Current state (Q_size,)
External Source
App. Input
-unpack tasks from network.
-add to queue
-update current state
Tasks transfer
Tasks Reception
.Concurent single threaded server arch.
-protocol: TCP
Network

Figure 3.7: Implementation architecture.

Chapter 4

Network Delays

The study of network delays gained attention lately when several services started using IP-based networks. These services include but are not limited to voice over IP (VoIP) [32] and teleoperation [31] that are significantly affected by delay variations and therefore require strict delay constraints. In systems where load-balancing is involved, delays greatly affect their stability in several aspects. First, the load distribution policies base their decisions on system state information that is outdated to a certain extent, and as the delay increases, the system becomes less stable. Moreover, this fact greatly affects the scalability of the system since the "error" in the global state information grows with the addition of new nodes. The use of prediction may not always be useful since network delays are unstable and vary according to several network conditions as will be shown in this chapter. Furthermore, fluctuations in the global system state also arise due to delays and variability in the transmission rates when load exchanges take place. In other words, migration of tasks between nodes may take an unknown amount of time. In fact, scheduled load distribution instances before the end of a task(s) transmission, causes the policy to base its assessment on old information i.e., the initial state of the system prior to the occurrence of the transmission. This fact when occurring frequently, renders global stability unachievable [22, 23, 24].

Moreover, the transition of tasks from one node to another may come at an unexpectedly high cost where the absence of transmission may have given better results. Therefore, a priori knowledge of the statistics of transmission delays may help the policy at hand wisely decide on the load distribution.

In this chapter, an experimental study of the transmission delays is introduced. In Section 4.1, Internet delays are investigated and categorized according to [11]. In Section 4.2, Local Area Network (LAN) delay probing experiments are presented. In Section 4.3 wireless network delays due to the transmission of mid-size TCP segments are investigated. Under high contention, TCP congestion control may exhibit chaotic behavior as shown in [44]. Nevertheless, the results of that section were used to model the delay distributions that were used in the stochastic model load-balancing simulator [20, 18].

4.1 Internet Delays

Nowadays, most delay experiments are performed on the RIPE NCC network, part of the Test Traffic Measurement (TTM) project [3]. RIPE NCC (Reseaux IP Europeen Network coordinator Center) [4] is a non-profit organization providing services for the benefit of the IP-based network operators (ISP) in Europe and the surrounding areas. The aim of the TTM project is to perform active measurements in order to detect the connectivity and to probe and monitor the one way delays between the different ISP networks.

In this section, some of the TTM delay experiments and their respective classes are presented. Then, delay probing experiments that we conducted over the Internet (not limited to Europe) are presented and compared to the different TTM categories.

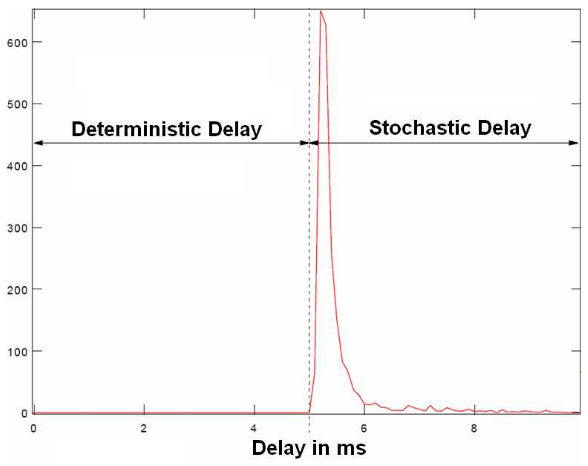


Figure 4.1: A typical delay histogram[11].

From end-to-end measurements, a typical delay is divided into two components, a deterministic and a stochastic one as illustrated in Figure 4.1. The deterministic delay includes the deterministic processing delay component, the transmission delay component, and the propagation delay component. The stochastic delay includes the stochastic processing delay and the queuing delay components. These different delay components are defined as follows,

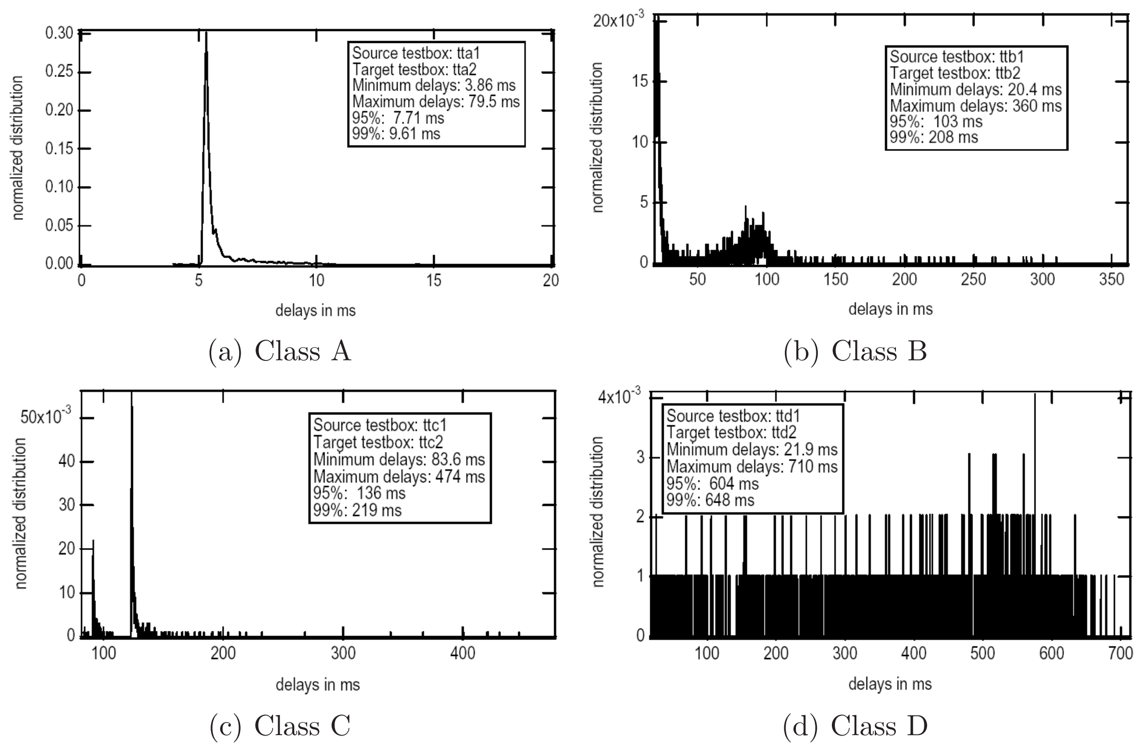
- **Processing delay** is the time needed to process a packet at a given node for transmission or reception. It has both deterministic and stochastic components due to variations in the node's computational power that affect the packet processing.
- **Transmission delay** is the time needed to transmit the entire packet. It depends on the bandwidth (link speed) and packet size.
- **Propagation delay** is the time needed to propagate one bit over the channel, and is primarily caused by the travel time of an electromagnetic wave. This delay is mostly observed in satellite links.
- **Queuing delay** is the storing time in routers along the path.

In [25] several methods were proposed to model the stochastic delays. They achieved an approximation of the processing delay distribution using a Gaussian pdf. Three parametric models were proposed for the stochastic queueing delay: the exponential model, the Weibull model, and the polynomial or Pareto model. All models exhibited discrepancies when they were compared to the available data.

On the other hand, Bovy et al. [11] classified the end-to-end delays into 4 categories. They used the RIPE one-way measurements with fixed IP probe-packets of 100 bytes. The configuration details are available in [21]. They characterized most of the experimental delay distributions as gamma-like distributions based on 2160 measurements taken per day per path. The 4 classes are listed below,

- **Class A** is the dominant and typical one and is modeled as gamma-like with a heavy tail that decays slower than an exponential. (Figure 4.2(a))
- **Class B** has a gamma-like shape with a Gaussian or triangular lob (Figure 4.2(b)). The second low-peak is due to changes in the network condition during part of the day.
- **Class C** has 2 gamma-like distributions due to a non-stable switch between 2 routing paths. (Figure 4.2(c))
- **Class D** has many peaks (white noise-like). This is mostly observed in paths that have high packet loss. (Figure 4.2(d))

We have performed planetary-scale delay probing experiments to investigate the different types of delay that may arise. Several planet-lab nodes were chosen to install our customized delay probing software. Round-trip measurements were used as opposed to the one way measurements adopted in the RIPE network. In fact, the TTM experiments were 10μs using GPS systems at each testbox. In our case,



(a) Class A (b) Class B

(c) Class C (d) Class D

Figure 4.2: Different classes of end-to-end delay distributions[11]

sufficient accuracy in clock synchronization was not available in Planet-Lab, hence we adopted the round trip time (RTT) in our experiments. The delay assessment of a certain path between node A and node B was performed as follows: Every 30s node A transmits a UDP packet of 50 bytes to node B. When node B receives it, it directly replies by sending another UDP packet of the same size to node A. This way, node A logs the difference in time between the transmission of the UDP packet and its corresponding packet reception. A 3s timeout is set by node A after which the packets are assumed to be lost or dropped by the network. Moreover, each probed path is monitored for 24 hours. Note that a UDP packet of 50 bytes results in an IP packet of 58 bytes (UDP header is 8 bytes) + 20 bytes (IP Header). This setup was used since the minimum official MTU (Maximum Transmission Unit) value allowed is 68 bytes [34]. Consequently, it is more likely that no fragmentation will be performed along the way. Additionally, the delay is calculated from the

Path	Nb. of trans.	Failed trans.	Error %	Min. delay (s)	Avg. delay (s)	Max. delay (s)	Std Dev.
UNM - Frankfurt	2809	18	0.64%	0.1012	0.2639	2.3136	0.126
Frankfurt - NTU (Taiwan)	2876	152	5.29%	0.3442	0.3642	0.6026	0.0246
INRIA (France) - Sinica (Taiwan)	2815	0	0.00%	0.1468	0.2937	0.5069	0.0248
NTU (Taiwan) - UNM	2850	71	2.49%	0.1217	0.2024	2.1751	0.0597
Arizona - MIT	2883	4	0.14%	0.0371	0.0815	2.1498	0.0797
Italy - France	2883	1	0.03%	0.0454	0.0575	0.8102	0.0355
Australia - London	2821	6	0.21%	0.1651	0.303	5.5802	0.1567
UNM - Australia	2857	6	0.21%	0.1323	0.2084	2.1575	0.0828
Hong-Kong - Canada	2848	2848	100.00%	-	-	-	-

Table 4.1: Summary of the delay probing experiments in the Internet

application layers' perspectives. That is, the time taken for the packet to travel through the TCP/IP protocol stack (upward and downward) is also included in the delay. This case is more relevant to the load-balancing system that is implemented in the application layer. The results for 8 different paths are summarized in Table 4.1. The resulting delay distributions based on measurements accumulated in 24 hours are shown in Figure 4.3.

At first sight, we can observe that our results are consistent with the RIPE experiments in the sense that the PDFs obtained are very similar in shape. Indeed, most of the distributions plotted in Figure 4.3 can be classified as class A. However, several triangular shapes were obtained that may not be well-modeled by a gamma distribution as is the case in Figures 4.3(d),(e),(g) and (h). Second, the distribution in Figure 4.3(a) has two peaks with one lower than the other which suggests that it belongs to class B. In fact, looking at the individual delay measurements plotted in Figure 4.4(a), we can see that between 11am and 5pm, higher network delays were present, which explains the shape of the corresponding distribution.

On the other hand, the path "France-Taiwan" exhibits a different behavior disclosed in its delay distribution in Figure 4.3(c). The pdf has an exponential rise followed by a sudden drop whereas in general, the inverse is observed i.e., the pdf

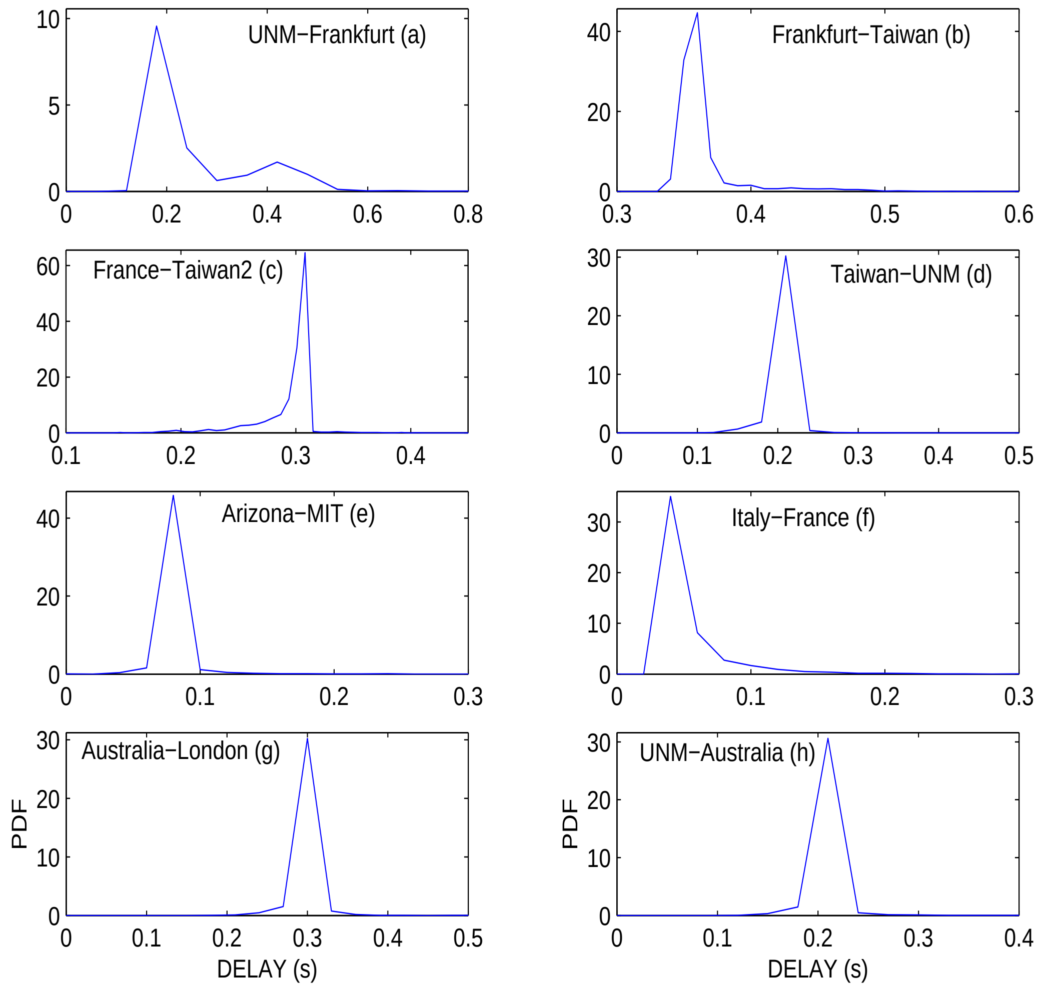


Figure 4.3: Delay distribution pdf for the different paths in the Internet (Taiwan is Sinica-Taiwan and Taiwan2 is NTU-Taiwan).

suddenly rises and then exponentially decays. This suggests further investigation by looking at the delay measurements (Figure 4.4(b)) and comparing them to a typical one (Figure 4.4(c)). We can observe that the delays measurements are clustered in the higher part of the plot whereas in Figure 4.4(c) the delays are clustered in its lower part. Thus, we can deduce that the link was mostly busy at that time which explains why such distribution is obtained. In general however, the typical distribution encountered is due to the fact that the links are lightly used or unsaturated.

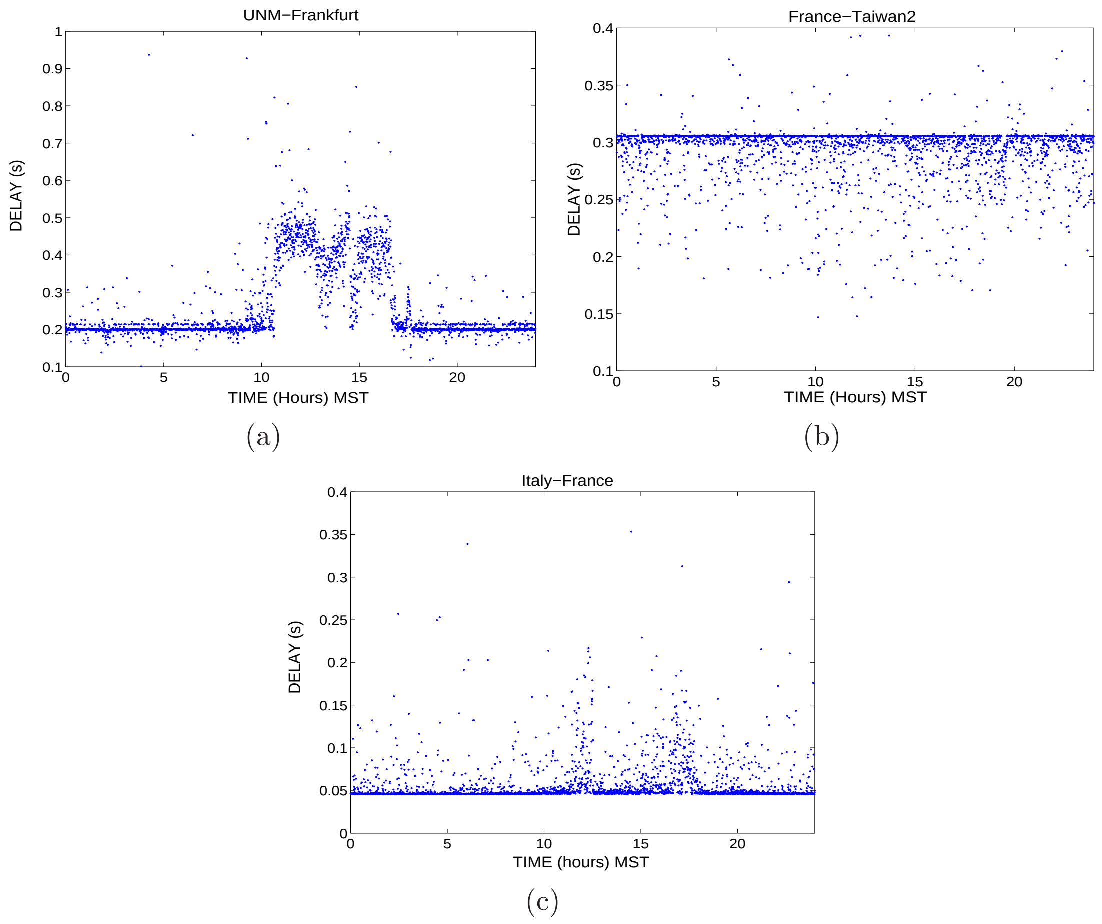


Figure 4.4: Individual delay measurements during 24 hours period (MST zone) for some paths on the Internet.

Consequently, this can also be used as a method to evaluate how busy a link is based on its delay distribution. Finally, although the two nodes available in Hong-Kong and Canada can be accessed from the University of New Mexico where the delay reports were collected, the two nodes were not able to reach each other in either direction. In fact, a *traceroute* run from the Hong-Kong node in direction of the Canadian node shows that the packet is dropped by the sixth hop in Hong-Kong, and a *traceroute* executed on the Canadian node in direction of the Hong-Kong node shows that the packet is dropped by the 8th hop also in Hong-Kong. This suggests that the Internet is not as completely connected as one would have thought.

4.2 Local Area Network Delays

The same delay probing experiment was performed on the ECE local area network. The two nodes picked were separated by at least 5 switches. The test was performed over 48 hours where 5754 measurements were collected. The minimum round-trip delay encountered was 317μs, the average delay 351μs, the maximum delay 1.12ms and the standard deviation 28.7μs. The delay distribution shown in Figure 4.5 is clearly a typical Class A sample.

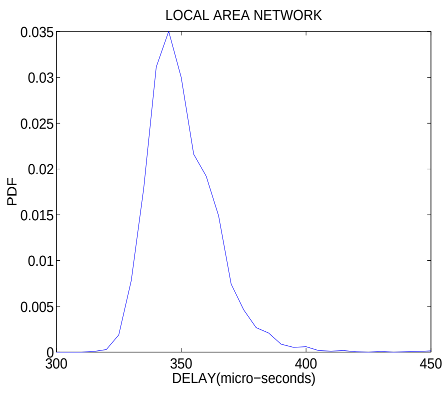


Figure 4.5: Delay distribution (pdf) for the ECE Local Area Network (LAN).

4.3 Wireless Network Delays

The wireless delay testing took a different aspect. Here, mid-size TCP segments of size 376KB were transmitted between the different nodes. Our main objective was to investigate the behavior of the tasks exchange part of the load-balancing system that runs over TCP. The delay distribution obtained was modeled and then integrated into the simulator used to validate the stochastic model. Load-balancing experiments performed on the wireless-testbed were compared later to the simulator results (Section 5.2).

From - To	Nb. of trans.	Failed trans.	Error %	Min. delay (s)	Avg. delay (s)	Max. delay (s)	Std Dev.
node1 - node2	449	18	4.0%	5.48	21.7	247.1	29.3
node2 - node3	1380	97	7.0%	3.703	8.3	29.3	3.1
node3 - node1	2446	19	0.8%	1.16	4.2	24	1.2

Table 4.2: Summary of the delay probing experiments in the wireless with AP network.

The delay probing from node A to node B was conducted as follows. Node A opens a TCP socket and writes on it 376KB of random data. When node B receives the entire segment, it sends back a 3 Bytes (ACK) acknowledgment on the same established connection. Actually, this scenario is the same when tasks exchange happens between two computational elements in the load-balancing system. Once again the delay is calculated from the application layer's perspective by taking the difference between the time the connection was established by node A and the time the ACK packet was received. This scheme was implemented on two different test-beds, an Ad-Hoc wireless network and a wireless network with infrastructure.

4.3.1 Wireless Delays in Ad-hoc Networks

The Ad-Hoc wireless test-best consists of 3 nodes connected amongst each other without the use of an AP (Access Point). The 3 nodes equipped with an 802.11b wireless adapter, were positioned inside the ECE department, where no direct line of sight was available between any 2 nodes. The three nodes were exchanging TCP segments at the same time for a period of 3 hours and 15 minutes. The results are summarized in Table 4.2. The individual delay measurements as a function of time and the corresponding pdf of each path are shown in Figures 4.6-4.8.

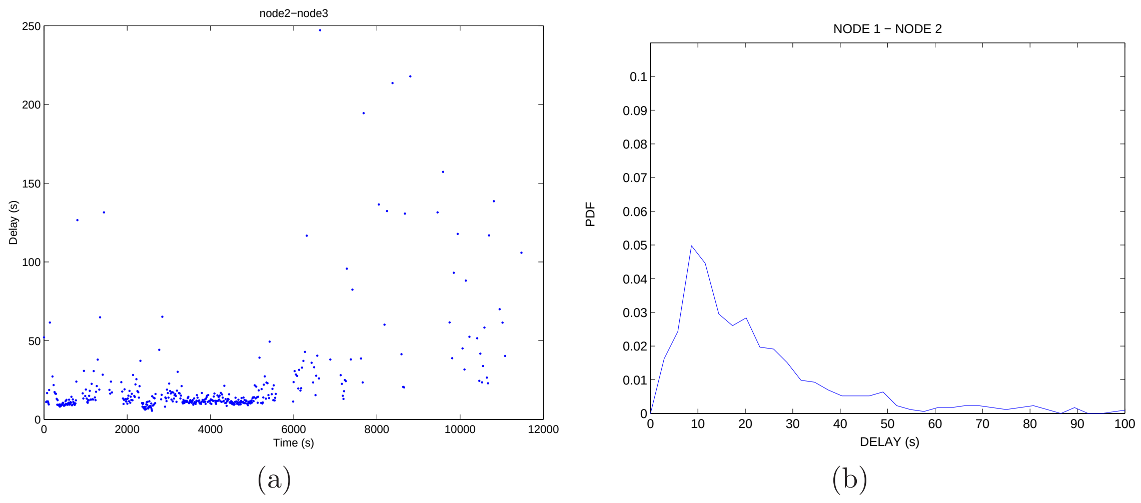


Figure 4.6: (a) Ad-hoc wireless network delay measurements between node 1 and node 2 for a 3 hours period as a function of time. (b) distribution of the delays for the same period.

It is clear that the wireless ad-hoc network exhibits a higher standard deviation and packet loss rate than the Internet which makes it less predictable. Moreover, as indicated by the path between node 1 and node 2, the wireless network is fragile in

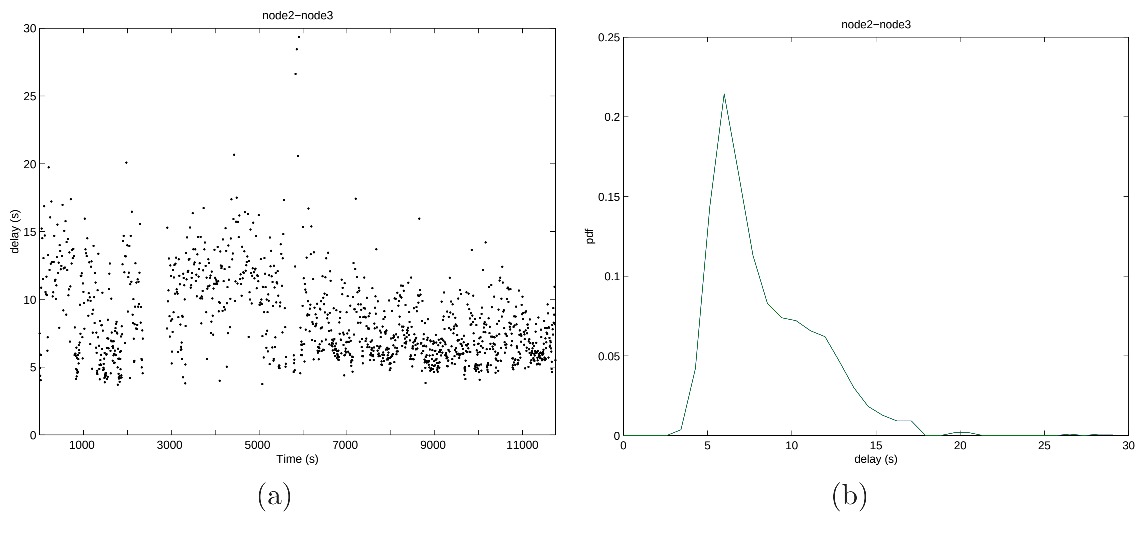
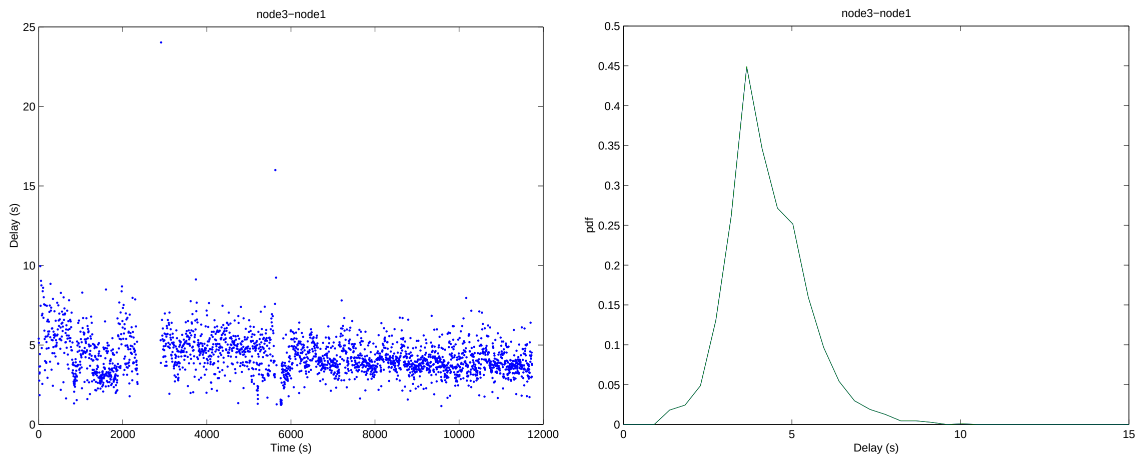


Figure 4.7: (a) Ad-hoc wireless network delay measurements between node 2 and node 3 for a 3 hours period as a function of time. (b) distribution of the delays for the same period.



(a) (b)

Figure 4.8: (a) Ad-hoc wireless network delay measurements between node 3 and node 1 for a 3 hours period as a function of time. (b) distribution of the delays for the same period.

the sense that it is affected by the slightest variation in the environment. This fact is indicated by the high standard deviation obtained (29.3) and mostly apparent in the sudden variation in the plot of Figure 4.6(a) which in turn explains the heavy tail of its corresponding distribution. Note that the nodes are stationary and therefore the main cause of such sudden delay variation could be related to any disturbance in the surrounding environment that has affected the wireless path between node 1 and node 2.

4.3.2 Wireless Delays in Networks with Infrastructure

The same delay probing experiments were performed on the ECE wireless network equipped with 802.11b access points (AP). Three nodes were also used and the setup of the experiment is shown in Figure 4.9. Each node was connected to a different AP located on a different floor in the ECE building where no possible signal interference could occur except from outside elements using the network (wireless or wired).The

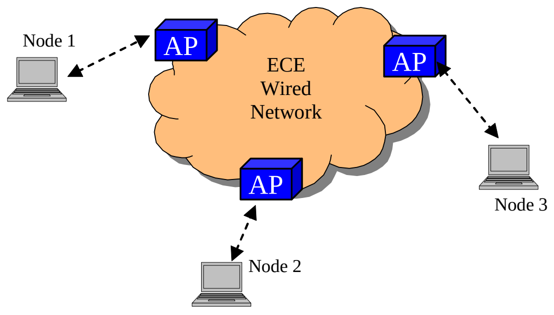


Figure 4.9: Setup of the delay probing experiment.

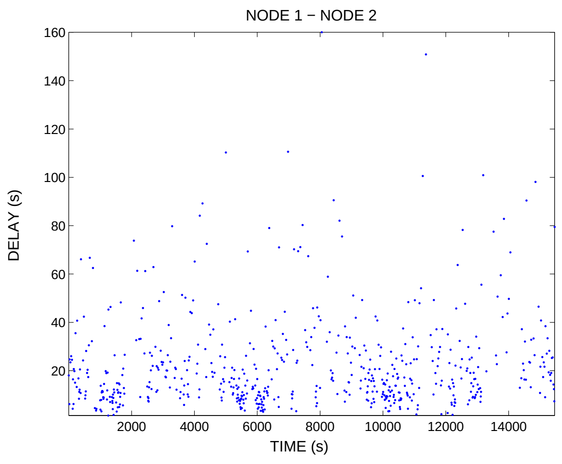
From - To	Nb. of trans.	Failed trans.	Error %	Min. delay (s)	Avg. delay (s)	Max. delay (s)	Std Dev.
node1 - node2	620	20	3.2%	1.4	22.6	160	19.9
node1 - node3	620	1	0.2%	3.029	7.6	37.4	2.7
node2 - node3	2172	18	0.8%	2.546	6.8	112.2	4.7
node3 - node1	1659	34	2.0%	2.357	9.2	35.6	4.5

Table 4.3: Summary of the delay probing experiments in the wireless with AP network.

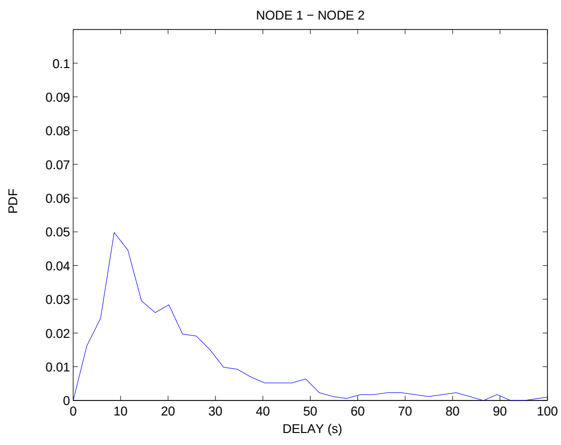
results are summarized in Table 4.3. The individual delay measurements as a function of time and the corresponding pdf of each path are shown in Figures 4.10-4.13.

Our first observation is that although the setup of the experiment seems symmetrical, no similarities between any of the paths can be found. Moreover, the path 1-3 did not exhibit the same characteristics in both directions. Nevertheless, the results shown in this section were better than the ones of the previous section in terms of packet loss and delay stability.

Plotting the different PDFs of the delays present in the wireless network on a log scale shows that most of them can be approximated by a straight line as shown



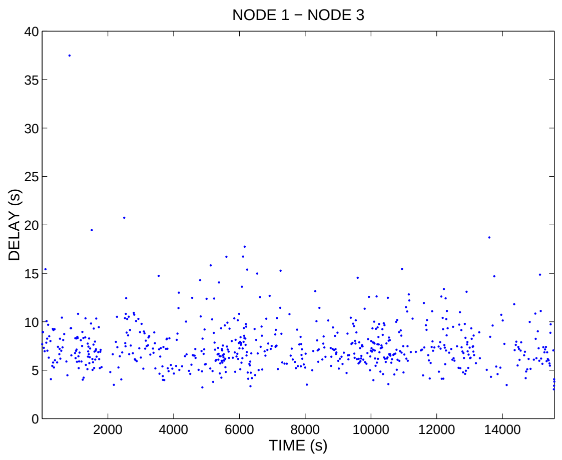
(a)



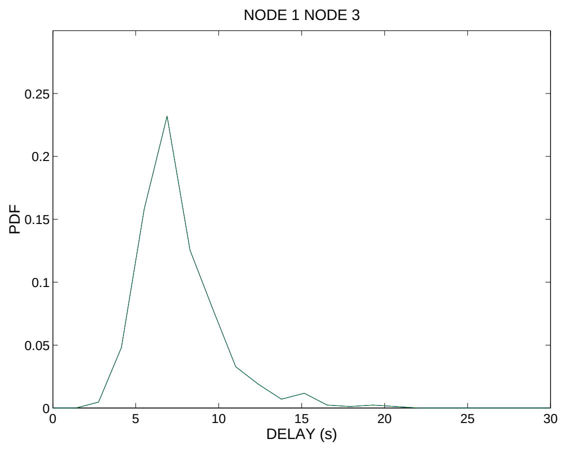
(b)

Figure 4.10: (a) Wireless with AP network delay measurements between node 1 and node 2 for a 4 hours period as a function of time. (b) distribution of the delays for the same period.

in Figure 4.14. Therefore, such delay distributions can be well approximated by exponential distributions with various slopes for use by the stochastic load-balancing simulator [18].

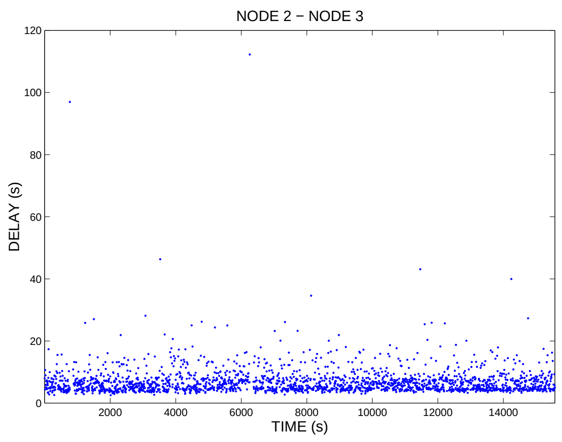


(a)

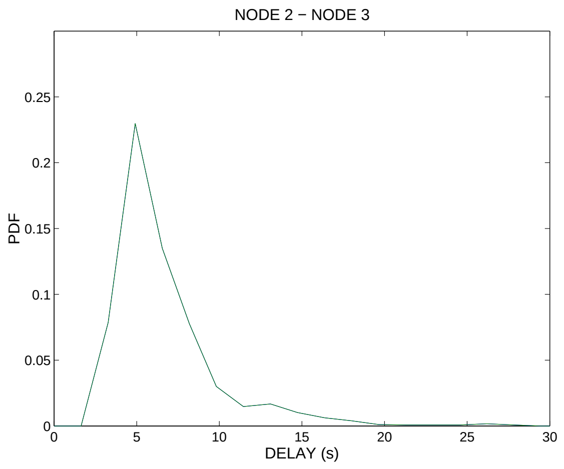


(b)

Figure 4.11: Wireless with AP network (a) delay measurements between node 1 and node 3 for a 4 hours period as a function of time. (b) distribution of the delays for the same period.



(a)

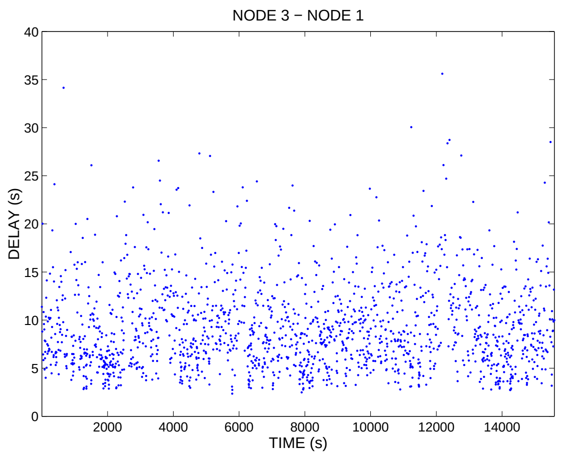


(b)

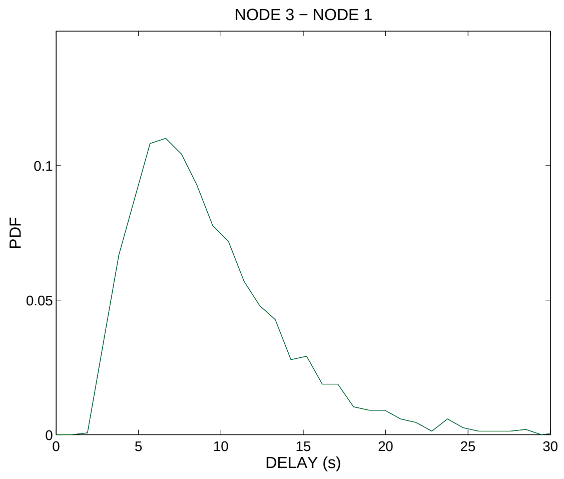
Figure 4.12: (a) Wireless with AP network delay measurements between node 2 and node 3 for a 4 hours period as a function of time. (b) distribution of the delays for the same period.

4.4 Summary

Delays in the Internet, LAN and wireless networks were investigated and categorized according to the shape of their probability density function. The four classes intro-



(a)



(b)

Figure 4.13: (a) Wireless with AP network delay measurements between node 3 and node 1 for a 4 hours period as a function of time. (b) distribution of the delays for the same period.

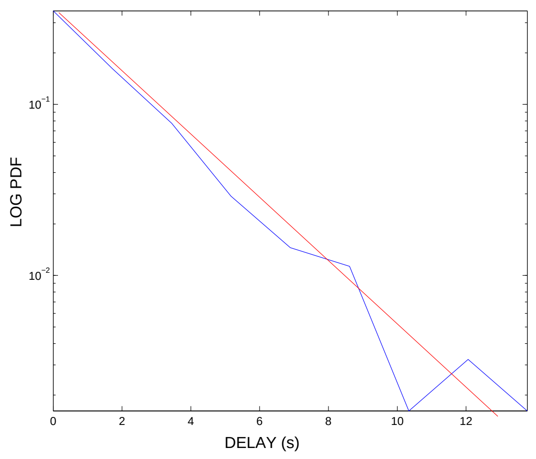


Figure 4.14: Example of a straight line fit for a shifted wireless delay pdf plotted on a logarithmic scale.

duced show that the delay is not predictable and varies greatly, which may affect systems where load-balancing is used. Moreover, connectivity between the nodes is not guaranteed; a link may become unavailable with higher probability in the Internet than in a LAN. In the wireless network, we noticed that the delay varied more frequently and packet drops were more prominent. Such delays can be approximated by an exponential distribution as shown when plotted on a log scale.

The delay probing experiments performed in this chapter will be helpful in understanding the behavior of the policies implemented in the load-balancing system of Chapter 3 and tested in different environments as will be seen in Chapter 5.

Chapter 5

Experimental results

In this chapter, experiments run on the load-balancing system described in Chapter 3 are presented. The main objective is to investigate the effect of network delays on the performance of the system. In fact, experimental optimization of the gain value K and the balancing instance t_b were performed. Analytical optimization work over t_b and K has been done by the load balancing group[1] at the University of New Mexico. In Section 5.1, the load-balancing policy with multiple balancing instances is considered when the fractions p_{ij} were set to a constant. First, we present the results of the experiments performed on a LAN test-bed at the University of Tennessee and then introduce the results of the experiments with the same settings completed on the Internet using Planet-Lab [2] nodes. In Section 5.2, a one-time load-balancing policy is considered where the fractions p_{ij} were set according to Equation (3.3). The experiments were done on the ECE wireless test-bed where experimental optimization over the one-shot balancing instance and the gain parameter K was performed. Moreover, the initial settings and the delay parameters resulting from the wireless experiments were incorporated into the stochastic model simulator. The

[1]Load Balancing Group at UNM website: http://www.eece.unm.edu/lb

results presented in this chapter were published in [22], [14], [18] and [23].

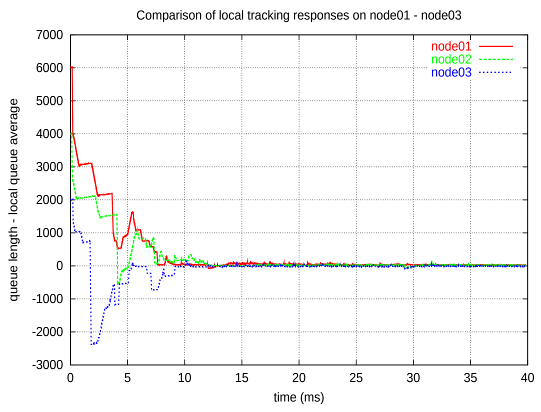
5.1 Multiple Balancing Instances

5.1.1 LAN Experiments

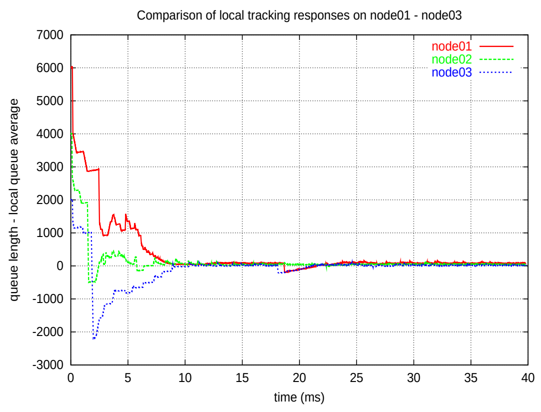
The experiments presented in this section were performed on 3 nodes connected by a switched network. The software system used was built at the University of Tennessee where the task is a query submitted to a search engine thread available on each node. The main interest here was to compare the experimental data with the simulations of the deterministic model (Section 2.3.1) that are available in [14]. Our ulterior objective however is to present these results for later comparison with the Internet experiments of the Section 5.1.2.

The initial settings of the experiment were as follow: The average time t_{p_i} to process a task is the same on all nodes (identical processors) and is equal to $10\mu\sec$ while the time it takes to ready a load for transfer is about $5\mu\sec$. The initial queue values inserted at each node are $q_1(0) = 6000, q_2(0) = 4000, q_3(0) = 2000$. node 1 was balancing every $75\mu s$, node 2 every $120\mu s$, and node 3 every $100\mu s$. All the experimental responses were carried out with constant $p_{ij} = 1/2$ for $i \neq j$.

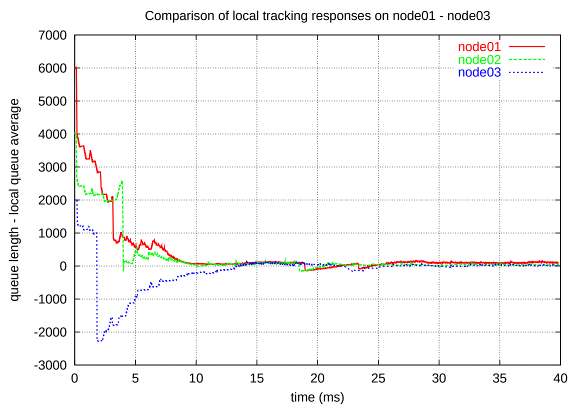
The plots of the system responses for different gain values K are shown in Figure 5.1. Figure 5.2 summarizes the data from several experimental runs of the type shown in Figures 5.1. For $K = 0.1, 0.2, 0.3, 0.4, 0.5$, ten runs were made and the settling time (time to load balance) were determined. These are marked as small horizontal ticks on Figure 5.2. (For all such runs, the initial queues were the same and equal to $q_1(0) = 600, q_2(0) = 400, q_3(0) = 200$. For each value of K, the average settling time for these ten runs was computed and is marked as a dot on given on Figure 5.2. For values of $K = 0.6$ and higher (with increments of 0.1 in K), consistent results could not be obtained. In many cases, oscillation extended throughout the experiment's time interval (200 milliseconds).



(a) the average gain value is $K = 0.5$.



(b) the average gain value is $K = 0.3$.



(c) the average gain value is $K = 0.2$.

Figure 5.1: Experimental response of the load-balancing algorithm. The plots show the excess load at each node versus time.

For example, Figure 5.3(a) shows the plots of the queue length less the local queue average for an experimental run with $K = 0.6$ where the settling time is approximately 7 milliseconds. In contrast, Figure 5.3(b) shows the experimental results under the same conditions where persistent ringing regenerates for 40 milliseconds. The response was so oscillatory that a settling time was not possible to determine accurately. However, Figure 5.2 shows that one should choose the gain close to 0.5

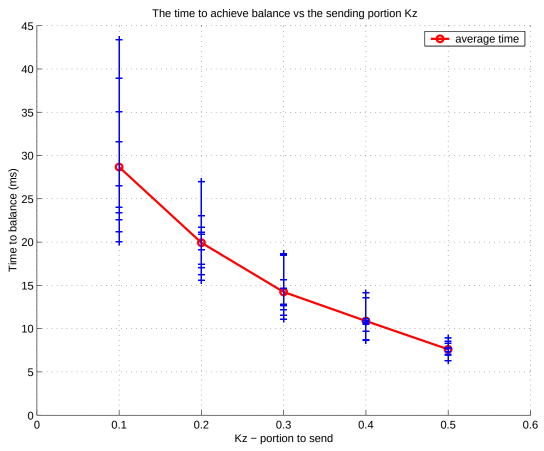


Figure 5.2: Summary of the load balance time as a function of the feedback gain K.

to achieve a faster response time without breaking into oscillatory behavior.

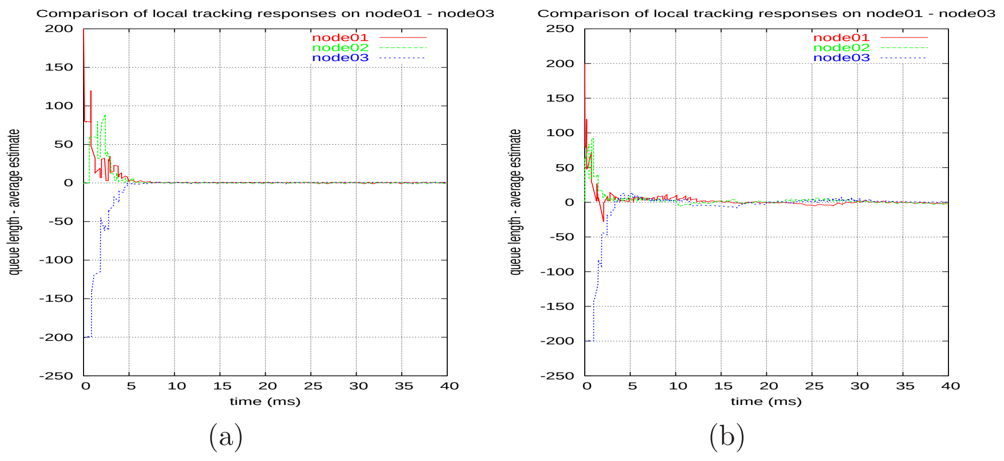


Figure 5.3: (a) $K = 0.6$ - Settling time is approximately 7 milliseconds. (b) $K = 0.6$ These are the same conditions as (a), but now the ringing persists.

5.1.2 Internet Experiments Over Planet-Lab

To match the experimental settings of the previous section, 3 Planet-Lab nodes were used; node1 at the University of New Mexico, node2 in Taipei-Taiwan and node3

	node 1	**node 2**	**node 3**
Location	University of New Mexico (US)	Taipei - Taiwan	Frankfurt - Germany
Initial Distribution	6000 tasks	4000 tasks	2000 tasks

Average Task Processing Time t_{pi}	10.2 ms
Standard Deviation for t_{pi}	2.5 ms
Interval between load balancing instances Δt	150 ms
Interval between 2 comm. transmissions	50 ms

Table 5.1: Parameters and settings of the experiment

	Roundtrip delay τ_{ij}	Data transmisison rate	Average Transmission of one Task
n1 - n2	215 ms	1.34 KB/s	14 ms
n1 - n3	200 ms	1.42 KB/s	16 ms
n2 - n3	307 ms	1.03 KB/s	20 ms

Table 5.2: Average network delays and transmission rates.

in Frankfurt-Germany. As for the load-balancing policy, the same parameter values were also used; for instance all p_{ij} were set to 1/2 for $i \neq j$. The initial parameters and settings for the experiment are summarized in Table 5.1.

Throughout the experiment, network statistics related to transmission rates and delays were collected. The averages of the parameters are shown in Table 5.2. Large delays were observed in the network due to the dispersed geographical location of the nodes. Moreover, the transmission rates detected between the nodes were very low mainly because the amount of data exchanged in bytes is small. Indeed, the average size of data needed to transmit a single task was 20 bytes, which made the observed transmission rates not exactly accurate in the presence of large communication delays.

As indicated previously, our interest is to compare the experimental results with the ones of the previous section and to assess the model under longer and more varying delays. In order to observe the behavior of the system under various gains, several experiments were conducted for different gain values K ranging from 0.1 to 1. Fig. 5.4 is a plot of the system responses corresponding to each node i where the gain K was set to 0.3. Similarly, Fig. 5.5 shows the system response for gain K equal to 0.5. Figure 5.6 summarizes several runs corresponding to different gain values. For each $K = 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,$ ten runs were made and the settling times (time to load balance) were determined. For gain values higher than 0.8, consistent results could not be obtained. For instance, in most of the runs no settling time could be achieved. However, when the observed network delays were less varying, the system response was steady and converged quickly to a balanced state when K is equal to 0.8 (Figure 5.7). As previously indicated, this scenario wasn't frequently seen. The system's behavior in these set of experiments does not exactly match, for the same gain value, the results obtained in the previous sections, due to the difference in network topology and delays. For instance, the ratio between the average delay and the task process time is 20 ($200\mu s/10\mu s$) for the LAN setting and 12 ($120ms/10ms$) for the distributed setting. This fact is one of the reasons why ringing is observed earlier (for $K = 0.6$) in the LAN experiment whereas under Planet-Lab unstable responses were observed starting which $K = 0.8$.

The previous experiments were conducted under normal network conditions stated in Table 5.2. However, another set of experiments was conducted at a different time where the network condition worsens and larger delays were observed. In particular, the data transmission rate between node 2 (Taiwan) and node 3 dropped from 1.03KB/s to 407B/s. Figures 5.8 and 5.9 show the system responses for gains $K = 0.4$ and $K = 0.8$ respectively. These experiments clearly show the negative effect of the delay on the stability of the system. Nevertheless, we can see that with a low gain namely $K = 0.4$, a settling time can be identified at around 22ms. On the

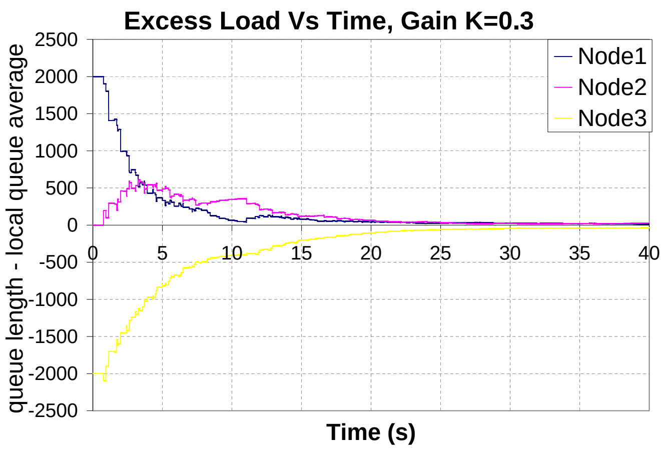


Figure 5.4: Experimental response of the load-balancing algorithm under large delays. gain $K = 0.3$ and $p_{ij} = 0.5$.

other hand, when the gain was set to 0.8, the system did not reach a stable point as shown by the nodes' oscillation responses in Figure 5.9. The reason behind this fact is that the load-balancing policy may base its decision on outdated information and consequently it becomes better not to migrate all the excess load.

As this point, only the effect of the delay on the stability of the system was

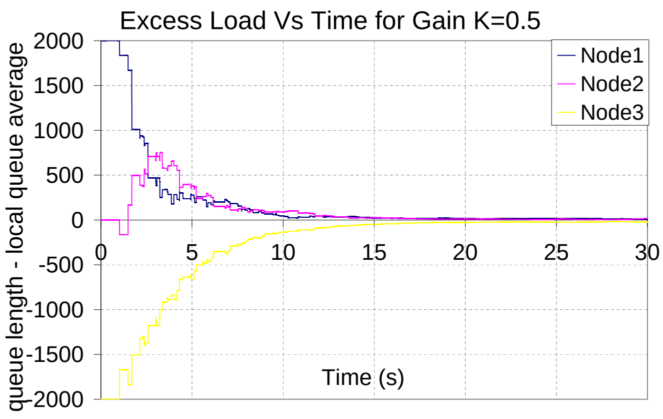


Figure 5.5: Experimental response of the load-balancing algorithm under large delays. gain $K = 0.5$ and $p_{ij} = 0.5$.

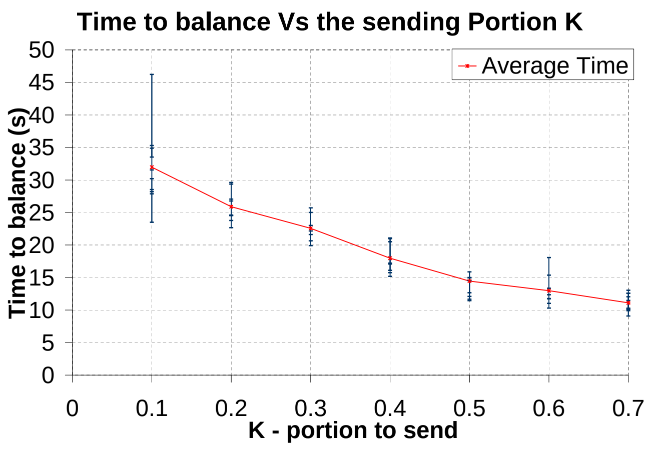


Figure 5.6: Summary of the load-balancing time as function of the gain K.

tested. In order to test the effect of the variability of the task processing time on the system behavior, the matrix multiplication application was adjusted in order to obtain the following results; the average task processing time was kept at 10.2ms but the standard deviation became 7.15 ms instead of 2.5 ms. This was done by adjusting the 2 parameters MAXBYTES and ROWSIZE introduced in Section 3.7. Figures 5.10 and 5.11 show the respective system responses for gains $K = 0.3$ and

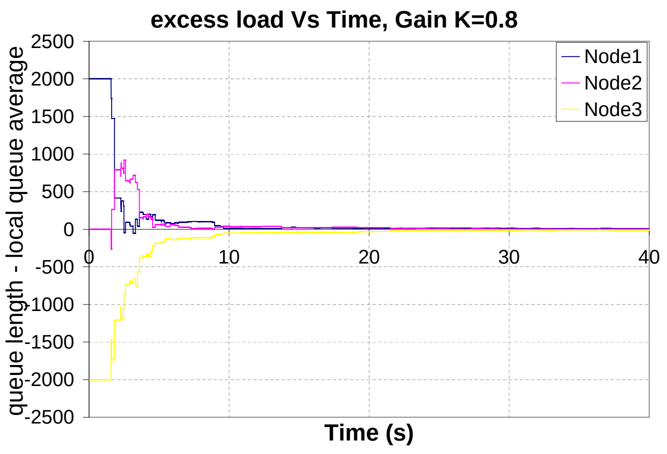


Figure 5.7: Experimental response of the load-balancing algorithm under large delays. gain $K = 0.8$ and $p_{ij} = 0.5$.

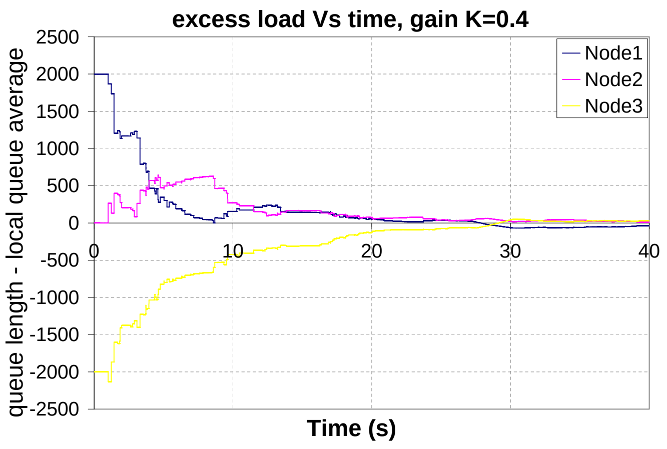


Figure 5.8: Experimental response of the load-balancing algorithm under large delays. gain $K = 0.4$ and $p_{ij} = 0.5$.

$K = 0.8$. Comparing Figures 5.4 and 5.10, we can see that in the latter case, some ringing persists and the system did not completely stabilize. On the other hand, setting the gain K to 0.8 led the system to accommodate the variances in the task processing time.

The results drawn from the two test-beds were consistent with each other. In

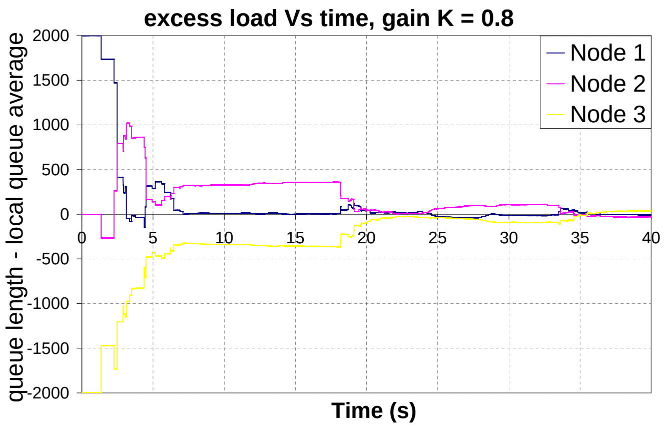


Figure 5.9: Experimental response of the load-balancing algorithm under large delays. gain $K = 0.8$ and $p_{ij} = 0.5$.

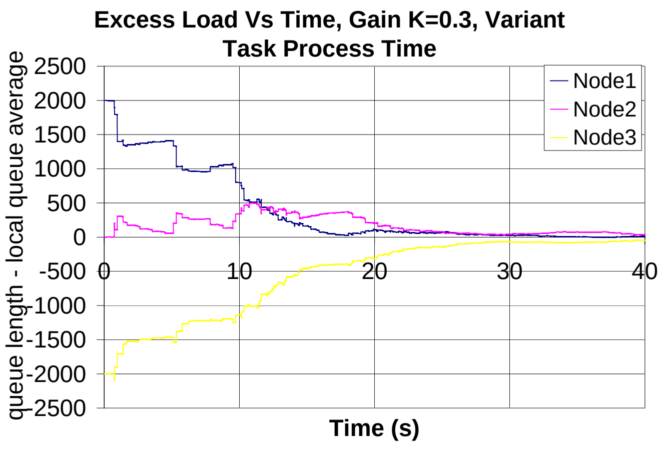


Figure 5.10: Experimental response of the load-balancing algorithm under large variance in the tasks processing time. gain $K = 0.3$ and $p_{ij} = 0.5$.

particular, high gains were shown to be inefficient and therefore introduce drawbacks in systems with large delays. Conversely, systems with low gain values could not cope with the variability introduced by the tasks processing time. Therefore, one should avoid the limiting cases and carefully choose an adequate gain value.

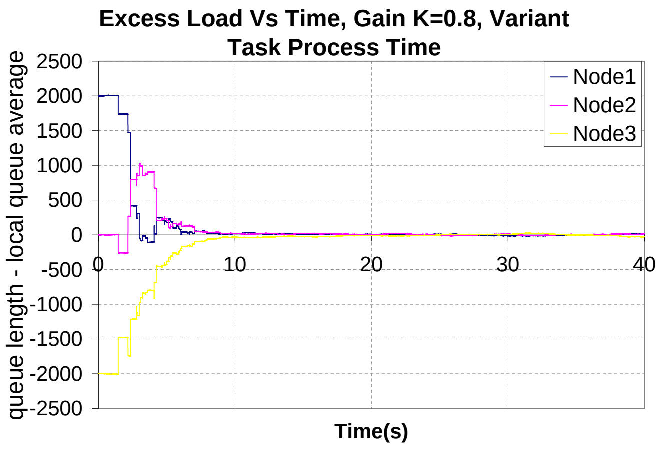


Figure 5.11: Experimental response of the load-balancing algorithm under large variance in the tasks processing time. gain $K = 0.8$ and $p_{ij} = 0.5$.

5.2 One-Shot Load-Balancing

In this section, load-balancing experiments were conducted over the wireless network where one load-balancing instance is chosen and the proportions p_{ij} were set according to (3.3) and is given as follows,

$$p_{ij} = \begin{cases} \frac{1}{n-2}\left(1 - \frac{Queue(i)}{\sum_{k=1,k\neq j}^{n} Queue(k)}\right) & \text{if all } Q(i) \text{ are known.} \\ 1/(\text{n-1}) & \text{otherwise} \end{cases} \tag{5.1}$$

5.2.1 Wireless Network Experiments

The experiments were conducted over a wireless network using an 802.11b access point. The testing was completed on three computers: a 1.6 GHz Pentium IV processor machine (node 1) and two 1 GHz Transmeta processor machines (nodes 2 & 3). To increase communication delays between the nodes (so as to bring the test-bed to a setting that resembles a realistic setting of a busy network), the access point was kept busy by third party machines, which continuously downloaded files. We consider the case where all nodes execute the load-balancing algorithm at a common balancing time t_b. On average, the completion time of a task was 525 ms on node 1, and 650 ms on the other two nodes.

The aim of the first experiment is to optimize the overall completion time with respect to the balancing instant t_b by setting the gain value K to 1. Each node was assigned a certain number of tasks according to the following distribution: Node 1 was assigned 60 tasks, node 2 was assigned 30 tasks, and node 3 was assigned 120 tasks. The information exchange delay (viz., communication delay) was on average 850 ms on the average. Several experiments were conducted for each case of the load-balancing instant and the average was calculated using five independent realizations for each selected value of the load-balancing instant. In the second set of experiments, the load-balancing instant was fixed at 1.4 s in order to find the optimal gain that minimizes the overall completion time. The initial distribution of tasks was as follows: 60 tasks were assigned to node 1, 150 tasks were assigned to node 2, and 10 tasks were assigned to node 3. The average information exchange delay was 322 ms and the average data transfer delay per task was 485 ms.

The results of the first set of experiments show that if the load-balancing is performed blindly, as on the onset of receiving the initial load, the performance is poorest. This is demonstrated by the relatively large average completion time (namely 45s$\sim$50s) when the balancing instant is prior to the time when all state communication between the CEs is completed (when t_b is approximately below 1s), as shown in Fig.5.12. Note that the completion time drops significantly (down to 40s) as t_b begins to approximately exceed the time when all inter-CE communications have arrived (e.g., when $t_b > 1.5$s). In this scenario of t_b, the load-balancing is done in an informed fashion, that is, the nodes have knowledge of the initial load of every CE. Thus, it is not surprising that load-balancing is more effective than the case the load-balancing is performed when the CEs have not yet received the state of the other CEs.

The explanation for the sudden rise in the completion time for balancing instants between 0.5 s and 1 s is that the knowledge states in the system are hybrid, that is, some nodes are aware of the queue sizes of the others while others arent. When this hybrid knowledge state is used in the load-balancing policy (Eqn. (6.1)), the resulting load distribution turns out to be severely uneven across the nodes, which in turn, has an adverse effect on the completion time. Finally, we observe that as t_b increases farther beyond the time all the inter-CE communications arrive (e.g., $t_b > 5$s), the average completion time begins to increase. This occurs precisely because any delay in executing the load-balancing beyond the arrival of the inter-CE communications time would increase the probability that some CEs will run out of tasks in the period before any transferred load arrives to them.

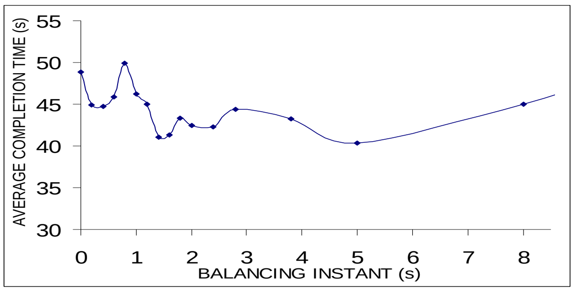


Figure 5.12: Average total task-completion time as a function of the load balancing instant. The load-balancing gain parameter is set at $K = 1$. The dots represent the actual experimental values and the solid curve is a best polynomial fit. This convention is used thought out Fig.5.15.

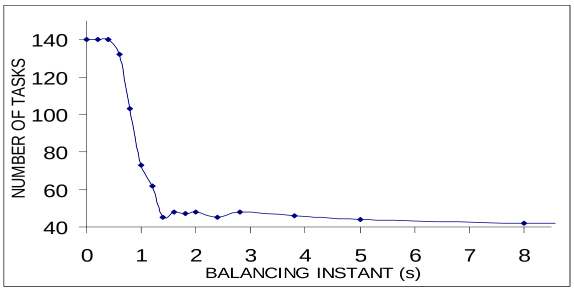


Figure 5.13: Average total excess load decided by the load-balancing policy to be transferred (at the load-balancing instant) as a function of the balancing instant. The load-balancing gain parameter is set at $K = 1$.

Next we examine the size of the loads transferred as a function of the instant at which the load-balancing is executed, as shown in Fig. 5.13. The illustrated behavior shows that the dependence of the size of the total load transferred on the "knowledge state" of the CEs. It is clear from the figure that for load-balancing instants up to approximately the time when all CEs have accurate knowledge of each other's load states, the average size of the load assigned for transfer is unduly large. Clearly, this seemingly "uninformed" load-balancing leads to the waste of bandwidth on the interconnected network.

The results of the second set of experiments indeed confirm our earlier prediction that when communication and load-transfer delays are prevalent, the load-balancing gain must be reduced to prevent "overreaction" (i.e., sending unnecessary excess load). This behavior is shown in Figure 5.14, and demonstrates that the optimal performance is achieved *not* at the maximal gain ($K = 1$) but when K is approximately 0.8. This is a significant result as it is unexpected in a situations where the delay is insignificant (as in a fast Ethernet case), $K = 1$ indeed yields optimal performance. Figure 5.15 shows the dependence of the total load to be transferred as a function of the gain. A large gain (near unity) results in a large load to be transferred, which in turn, leads to a large load-transfer delay. Thus, large gains increase the likelihood of a node (that may not have been overloaded initially) to complete all its load and remain idle until the transferred load arrives. This would clearly increase the total average task completion time, as confirmed earlier by Fig. 5.14.

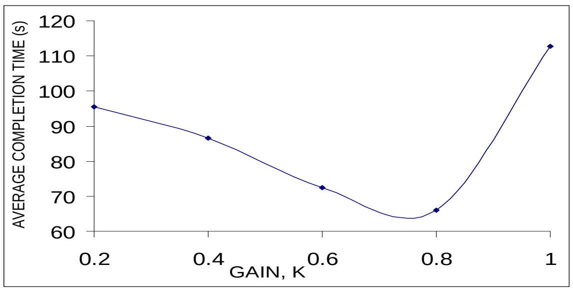


Figure 5.14: Average total task-completion time as a function of the balancing gain. The load-balancing instant is fixed at 1.4 s.

.

5.2.2 Simulation Results

A Monte-Carlo Stool that allows the simulation of the queues described in Section 2.3.2 was developed at the University of New Mexico [20]. The network parame-

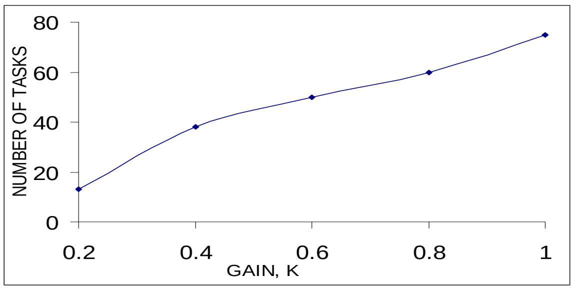


Figure 5.15: Average total excess load decided by the load-balancing policy to be transferred (at the load-balancing instant) as a function of the balancing gain. The load-balancing instant is fixed at 1.4 s.

ters (i.e., the statistics of the communication delays η_{kj} and the load transfer delays τ_{ij}) and the task execution time in the simulation were set according to the respective average values obtained from the experiments described in the previous section. This simulation tool was used to validate the correspondence between the stochastic queuing model and the experimental setup. In particular, the simulated versions of Figures 5.12 –5.14 were generated, and are shown in Figures 5.16–5.18.

It is observed that the general characteristics of the curves are very similar, but they are not exactly identical, due to the unpredicted behavior and complexity of the wireless environment. Nevertheless, the result of the first simulation, shown in Fig.5.16, were consistent with the experimental result as we can clearly identify the sudden rise in the completion time around the balancing instant corresponding to the communication delay (850 ms). The reason for this behavior was described in the experimental section. As for the excess transferred load plotted in Fig.5.17, the simulation resulted in the same curve and transition shape obtained from the experiment. The curve characteristics of the second simulation, shown in Fig. 5.18, are also analogous to the ones obtained in the experiment. Indeed, the gain values found are almost the same: 0.8 from the experiment and 0.87 from the simulation. As indicated before, the small difference is due to the unstable delay values and other factors present in the wireless environment, which have been approximated both by the model and the simulator.

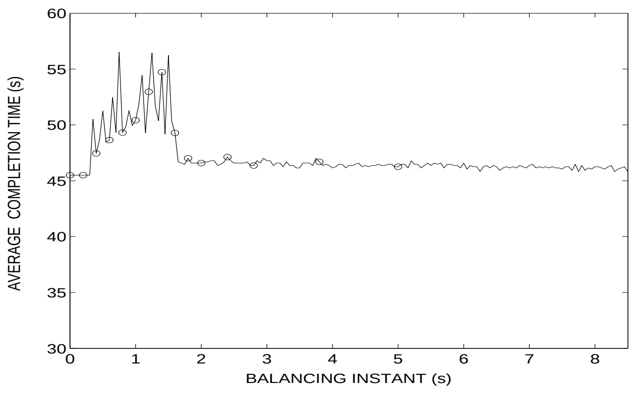


Figure 5.16: Simulation results for the average total task-completion time as a function of the load-balancing instant. The load-balancing gain parameter is set at $K = 1$. The dots represent the actual experimental values[20].

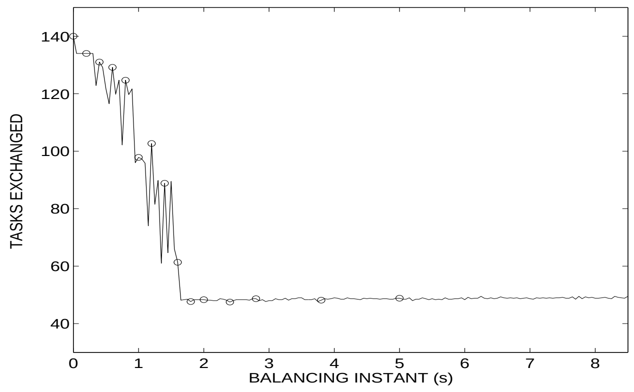


Figure 5.17: Simulation results for the average total excess load decided by the load-balancing policy to be transferred (at the load-balancing instant) as a function of the balancing instant. The load-balancing gain parameter is set at $K = 1$ [20].

5.3 Summary

In this chapter, experimental results of the load-balancing system were presented in two different environments; the Internet (Planet-lab) and the wireless network. In the

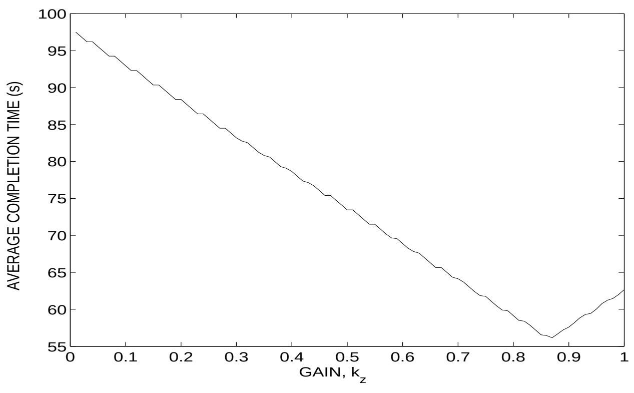


Figure 5.18: Simulation results for the average total task-completion time as a function of the balancing gain. The load-balancing instant is fixed at 1.4 s. [20]

Internet, the results showed that a gain parameter is necessary to compensate for the delay incurred in the transfer of information and the variability in the computational power at each node. As predicted by the stochastic model, the monte-carlo simulation and then shown in the experimental section, a gain K value of 1 does not give optimal result. Therefore, one should set K the mid range of $(0, 1]$ in systems where network delays are prominent.

In the wireless network, our experimental results and simulations both indicate that in systems where communication and load-transfer delays are tangible, it is best to execute the load-balancing after each node receives information from other nodes regarding their load states. In particular, our results indicate that the loss of time in waiting for the inter-node communications to arrive is compensated for by the informed nature of the load-balancing. For both systems, the optimal load-balancing gain turns out to be less than unity, contrary to systems that do not exhibit significant latency. In delay-infested systems, a moderate balancing gain has the benefit of reduced load-transfer delays, as the fraction of the load to be transferred is reduced.

Nevertheless, the policies implemented so far do not account for load transfer delays and connectivity in the system in a direct way but only through the use of a gain parameter. In Chapter 6, an adaptive load-balancing policy is introduced to probe the system and uses its performance history to decide on an adequate load distribution.

Chapter 6

Dynamic and Adaptive Load-Balancing Policy

In this chapter, a new dynamic, adaptive, and distributed load-balancing policy is introduced based on the policies that were used earlier in this thesis. This strategy takes into account the following three aspects: i) the connectivity among the nodes, ii) the computational power of each node, and iii) the throughput and bandwidth of the system.

The occurrence of load-balancing instances, i.e.when to trigger the load distribution strategy, is not discussed in this chapter since it is possible to employ one-shot load-balancing, multiple balancing instances, or any other scheme that would be suitable for the system at hand. Note that experimental (Section 5.2) and theoretical studies [18] have been done in the optimization of the load-balancing instances and dynamic strategies can be used based on system events as described in [20]. Furthermore, exchanged information about the state of the nodes is still assumed to occur frequently as described in previous chapters, with additional information to be discussed subsequently. In Section 6.1, The new load distribution strategy is introduced and in Section 6.1.1, the computational methods of the adaptive parameters used by the policy are described. Section 6.1.2 presents an experimental evaluation of this new dynamic policy.

6.1 Dynamic and Adaptive Policy Description

The Internet is not as completely connected as one might think. This has been observed in Section 4.1 where nodes in Hong-Kong and Canada weren't able to reach each other although they were perfectly accessible from other sites such as UNM. Add to that fact that in distributed systems, a node may become unavailable or unreachable at any time due to a failure in the node itself, or in the network path leading to it. Therefore, the assumption made by the load-balancing policies that all nodes are accessible at any time is unrealistic especially in Internet scale distributed systems or in Ad-Hoc wireless networks. This will greatly affect the load balance state of the system since loads assigned to unreachable nodes can never be delivered. The proposed algorithm can detect the connectivity in the system and decide accordingly what nodes may participate in the load sharing. At each load-balancing instance, the group of reachable nodes is referred to as the "current node space".

This load-balancing policy also takes into account the change in the computational performance of nodes and distributes the tasks accordingly. Actually, the system is not dedicated to the application at hand; other users may be using one or more nodes at a given time and therefore alter their computational power. Moreover, tasks are not considered identical, they may greatly differ in their completion time. These facts cannot be guessed a priori and assigning fixed computational power for each node is not always suitable. Therefore, the load strategy should be adaptive to these changes and be able to make decisions correspondingly.

Moreover, transfer delays incurred when tasks are migrated from a node to another may be unexpectedly large and result in a negative impact on the overall system performance. To avoid this situation, an a priori estimate of the transfer delays will help the policy at hand decide if the transfer is profitable and adequately decide on the size of load to migrate. These estimates should also be dynamically updated since delays may greatly vary during the system's life as shown for delays of class B and higher in Section 4.1.

To further describe the policy, the following parameters are defined.

- n is the number of nodes present in the system.
- q_i denotes the number of tasks in the queue of node i.
- C_i denotes the average completion time of a task at node i. Without loss of generality, we will assume that C_i is in seconds.
- s_i is the average size of a task in bytes at node i when it is transferred.
- r_{ij} is the throughput or transfer rate in bytes/seconds between node j and node i. Note that $r_{ij} \neq r_{ji}$.
- $q_{j,av}$ is the average queue size calculated by node j based on its locally available information.
- $q_{j,excess}$ is the excess number of tasks at node j.
- p_{ij} is the fraction of the excess tasks of node j that should be transferred to node i as decided by the load balancing policy.

The first 5 parameters are assumed known at the time the load distribution process is triggered. That is, the update of these variables is done before the balancing instance is reached as will be described later. The general steps of the load-balancing policy invoked at node j are described below followed by a detailed description.

1. ***Determine*** how many nodes are reachable (n_o) from node j.

   Calculate the average queue size $q_{j,av}$ and the number of excess tasks $q_{j,excess}$ based on locally available information.

   $$q_{j,av} = \frac{1}{n_o} \sum_{i=1,\ \text{node}(i)\ \text{reachable}}^{n} q_i \frac{C_i}{C_j}$$

   $$q_{j,excess} = \begin{cases} (q_j - q_{j,av}) * K & \text{if } q_j > q_{j,av}, \\ 0 & \text{Otherwise,} \end{cases}$$

   where K is the gain parameter.

   The algorithm ***exists*** if $q_{j,excess} = 0$.

2. ***Determine*** how many n'_o and which nodes are below the average. These nodes will participate in the load sharing as viewed by node j.

3. ***Calculate*** the optimal p'_{ij} fraction only for the n'_o nodes using the following formula:

   $$p'_{ij} = \frac{q_{i,av} - (C_i/C_j)q_i}{\sum\limits_{k\ \ni\ k \neq j} q_{k,av} - (C_k/C_j)q_k}$$

4. ***Calculate*** the p''_{ij} for the n'_o nodes. p''_{ij} is the maximum portion of the excess load that is judged to be profitable when transmitted to node i.

   $$p''_{ij} = \frac{(q_j - q_{j,excess})C_j r_{ij}}{q_{j,excess} s_j}$$

   Set $p_{ij} = Min(p'_{ij}, p''_{ij})$.

5. *if* $\sum_i p_{ij} > a$ (a is a threshold parameter between 0 and 1)

   transmit $(p_{ij} q_{j,excess})$ tasks to node i,

   Otherwise

   repeat step 1 to step 4,

```
    assign the remaining fraction $(1 - \sum_i p_{ij})$ to nodes that have $p''_{ij} > p'_{ij}$
    and call the newly assigned fractions $p'''_{ij}$,
    transmit $(p_{ij} + p'''_{ij})q_{j,excess}$ tasks to node $i$.
```

The first step determines the node space where the load distribution will take place from node j's perspective. This is achieved by checking the last time a state or SYNC packet was received from each node. To test for connectivity to node i, the `local_timestamp` (in the *info* structure, Section 3.3) of the last received state packet is compared to the current time decremented by three times the interval between 2 consecutive state broadcasts. That is, if the last state packet received from node i is one of the last three packet transmitted, node i is considered to be reachable from node j otherwise it is not included as part of the load-balancing node space since most likely, a load transmitted to this node will not be correctly delivered. Therefore, it is more suitable for the policy to base its calculations on nodes where migration of loads have higher probability od success. Note that, at every instance of the load distribution process, the node space may end up with different elements according to the nodes' connectivity at that time. After that, the local queue average is calculated where each queue is scaled by the C_i/C_j factor that accounts for the difference in computational power of each node. In the queue excess calculations, a gain factor K is used since it is becoming a requirement in any policy that operates on large-delay systems where outdated state information is prominent. This fact appears in the experimental and theoretical studies conducted in the literature and in this thesis.

The second and third steps employ the method introduced in Section 3.8 and used earlier in the literature in the SID and RID policies (Section 2.4) to calculate the fractions p'_{ij}. This method is attractive since it only considers nodes that have queue sizes less than the average, and therefore results in as few connections as possible when the excess load is moved out of node j which makes the policy at hand more scalable (Figure 3.5). Moreover, this method leads to optimal results when the policy is triggered once on each node and where no delay is present in the system. Note that the formula is adjusted by the C_i/C_j factor.

The fourth step judges if the proportion p'_{ij} determined is worth transmitting to node i when transmission delays are present. This is accomplished by setting an upper bound on the maximum proportion of the excess load that is profitable when the exchange takes place. The task migration is said to be profitable if the time needed to transmit the load to the other end is less than the time needed to start executing the load on the current node (node j in this case). This statement is interpreted as follows,

$$\underbrace{p''_{ij} q_{j,excess}(s_j/r_{ij})}_{\text{transmission delay}} < \underbrace{(q_j - q_{j,excess})C_j}_{\text{start of load execution}}$$

Solving for p''_{ij}, we get the upper bound for p_{ij} as indicated in step 4. In case r_{ij} is not available, r_{ji} is used instead to provide an approximation of the bandwidth between node j and i. If neither parameter is available, step 4 is omitted for the node pair (i, j). The rate r_{ij} is detected and updated each time a load is transmitted from node i to node j as will be explained in the subsequent section where r_{ji} is received in the state information packet transmitted by node i to node j. Both parameters are saved respectively in the variables `rate` and `symm_rate` in the *info* structure (Section 3.3).

The fifth step is included for completion and can be omitted at any time. The rational behind it, is that after executing the algorithm, node j may find itself only transmitting a small portion (i.e less than a variable a) of its excess load due to the delay restrictions. Therefore, it is suitable to reassign the remaining untransferred proportion to the nearest nodes (i.e nodes reached through links of higher rate) in the hope that they may possibly have better connectivity to the system.

6.1.1 Adaptive Parameters Computation

In this section, the computation procedure for the dynamic parameters C and r_{ij} is explained. Note that the s_i parameter can be easily determined by averaging the tasks' sizes upon their creation.

Every time a task is completed by the application layer at node i, the C_i parameter is updated as follows,

```
if $C_i = 0$ then $C_i = T_{task}$
else $C_i = \alpha T_{task} + (1 - \alpha) C_i$
```

where T_{task} is the execution time of the last task and α is a gain parameter that affects the C_i term in its ability to reflect the current computational power of the system. Therefore, the values of α are critical to the stability and efficiency of the load-balancing policy. That is, assigning values in the high range of (0,1] to α may result in fluctuations in the C_i parameter which will have in turn an adverse impact on the decision of the load distribution, leading to bouncing of tasks back and forth between nodes. On the other hand, setting α to low values may not keep the load-balancing policy informed about the latest state of the node. Consequently, the value of α should be selected depending on the application used and the interference degree of external users. The update procedure could be easily modified to suit other methods.

The other parameter that is dynamically updated is the transfer rate r_{ij} incurred between node j and node i. On each data (or tasks) transmission, the transfer delay T_{delay} is recorded and is calculated by taking the difference between the instance the connection is initiated by node j and the instance node i acknowledgment of tasks reception is received by node j. Consequently the average transmission rate ($rate = T_{delay}/totalsize$) is calculated, where $totalsize$ is the total size of the tasks migrated to node i. After each successful exchange of loads, the r_{ij} parameter is updated as follows,

```
if $r_{ij} = 0$ then $r_{ij} = rate$
else $r_{ij} = \beta * rate + (1 - \beta) r_{ij}$
```

This scheme is a simplified version of the method used to update the Round Trip Time (RTT) of the packet exchanged during a TCP connection where the delay variance is additionally taken into consideration [26]. In the next section, β was set to 1/8 as suggested by [27] for the RTT update method.

Finally, both parameter C_j and r_{ij} are included in node j state information when transmitted to node i for all $i = 1, ..., n$, $i \neq j$.

6.1.2 Experimental Evaluation

To test the performance of the newly proposed load-balancing policy denoted as lb2, a comparative experiment was performed between this policy and the strategy adopted by the stochastic model introduced in Section 3.8 and referred to as lb1. The experiments were conducted over Planet-Lab [2] and the initial settings and parameters are shown in Table 6.1. The average network transfer rates for each path as detected by the system are shown in Table 6.2. The round trip end-to-end delay values were consistent with the results obtained in Section 4.1 and were shown in Table 4.1.

First, lb2 was evaluated for the gain values K between 0.3 and 1 with 0.1 incremental steps. The α parameter introduced in the previous section was set to 0.05 by running several experiments and observing the behavior of the C parameter. Note that, the first time the load-balancing process was triggered was after 20s from the start of the system and then the strategy was executed regularly at 10s intervals.

	node 1	**node 2**	**node 3**
Location	UNM (University of New Mexico)	Frankfurt - Germany	Sinica - Taiwan
Initial Distribution	600 tasks	250 tasks	100 tasks
Average Task Processing Time *C* (ms)	160	400	500

Average size of a task (Kbytes)	3.12
Interval between 2 load balancing instances (s)	1.5
Interval between 2 state transmissions (s)	10

Table 6.1: Parameters and settings of the experiment.

From - To	**node 1**	**node 2**	**node 2**
node 1	-	34.5 KB/s	73.3 KB/s
node 2	18.7 KB/s	-	45.4 KB/s
node 3	48.9 KB/s	20.2 KB/s	-

Table 6.2: Average transmission rates between the different nodes.

This was done to ensure that the C parameter had enough time to adapt and reflect the current computational power of each node before the occurrence of any tasks migration between the nodes.

Second, lb1 was evaluated under the same conditions as lb2. Since, there is a discrepancy between the computational power of the nodes (as shown in Table 6.1), the lb1 strategy was adjusted to account for these differences by scaling the queue sizes in the p_{ij} computation as follow,

$$p_{ij} = \begin{cases} \frac{1}{n-2}\left(1 - \frac{(C_i/C_j)Queue(i)}{\sum_{k=1,k\neq j}^{n}(C_k/C_j)Queue(k)}\right) & \text{if all } Q(i) \text{ are known.} \\ 1/\text{(n-1)} & \text{otherwise} \end{cases} \tag{6.1}$$

Note that the ratio C_i/C_j are fixed over time. Their values were obtained from the lb2 experiments by averaging over all the tasks processing time at each node.

Both policies were evaluated by conducting 5 runs for each value of K between 0.3 and 1 with 0.1 incremental step. Figure 6.1 shows the overall average completion time versus K and Figure 6.2 shows the total number of exchanged tasks between all the nodes.

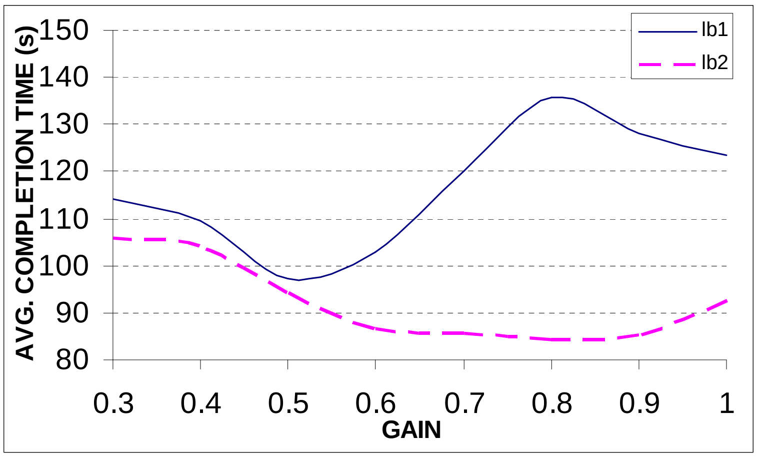


Figure 6.1: completion time averaged over 5 runs Vs different gain values K. The graph shows the results for both policies.

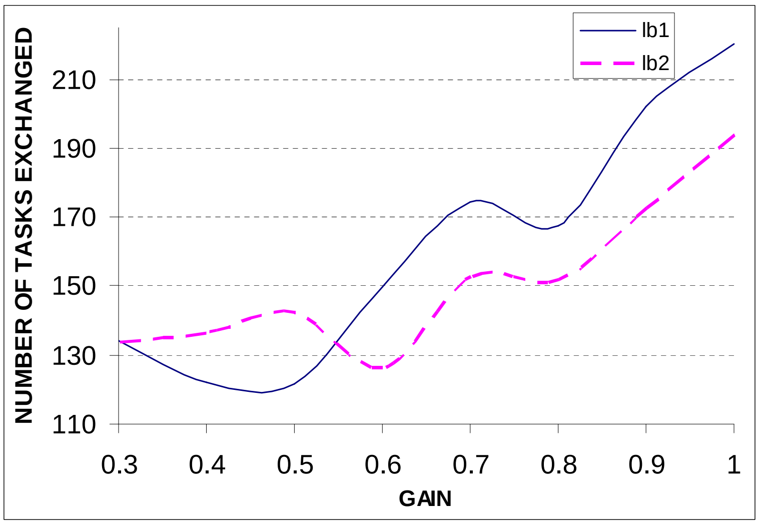


Figure 6.2: Total number of tasks exchanged averaged over 5 runs Vs different gain values K. The graph shows the results for both policies.

We can clearly see that lb2 outperformed lb1 especially for $K = 0.8$ that corresponds to lb2 earliest completion time whereas lb1 performed best at $K = 0.5$. The reason may be that lb1 used greater predictive computations before distributing the loads, which makes it "more or less" independent of the gain value K (in the range [0.60.9]). As for the network traffic generated during the lifetime of the system, lb2 had fewer tasks exchanged for most of the gain K values. It is expected that the difference in total tasks migrated between the 2 policies will grow as the number of nodes increases.

In this experiment, few aspects of the lb2 policy were examined. Further tests should be performed under different conditions; a higher number of nodes (to test scalability), bigger tasks size (to investigate the effect of transfer delays). Moreover, theoretical and experimental studies should be done in the optimization of the α and β parameters.

6.2 Summary

In this chapter, a dynamic load-balancing policy is introduced to take into account the connectivity between the nodes, the variability in computational power and the network transfer delays. Preliminary experimental results show that the proposed strategy provides improvements over policies implemented earlier in this thesis. More work should be done in investigating its capabilities and implementation enhancement.

Chapter 7

Conclusions and Future Work

In this thesis, we first presented a brief description of the different taxonomies of load-balancing policies followed by an overview of previous work in the field. We then presented the design and implementation of a general framework where distributed load-balancing policies can be tested and compared. The multi-threaded architecture of the system provides high performance for the application at hand and is not halted when transfer of information or loads takes place. We also showed that the system provides flexibility in integrating different types of distributed, dynamic, and adaptive strategies.

Delay probing experiments performed on the Internet and the wireless test-beds (both ad-hoc and with infrastructure) showed that these environments are unstable in the sense that high network delay variabilities and packet drops were observed especially in the wireless network. Furthermore, connectivity between nodes in the Internet is not always guaranteed. These facts had a considerable influence on the performance of the implemented strategies in these different environments. The gain parameter K was found to have a great impact on the stability of the system where a value in the mid range of the interval $(0, 1]$ provided the best results. Moreover, load-balancing should always occur in an informed manner directly after the receipt of the state information from all the nodes belonging to the current balancing space.

Based on the delay probing experiments and the performance of the dynamic load-balancing strategies in different environments, we proposed a dynamic and adaptive load-balancing policy that takes into account the connectivity in the network, the variability in the transfer delays, and the computational power of each node. Preliminary experimental results show that this policy provides improvements over the previously implemented dynamic strategies.

In future work, we will provide improvements for the load-balancing software in order to let any application run in a distributed manner and benefit from a wide range of implemented load-balancing policies. This will be accomplished by integrating a set of API (Application Program(ming) Interface) functions in the software package through which any application will be able to communicate with the two lower layers (load-balancing and network communication). Furthermore, the application will be separated from the software package in the sense that it will have its own execution space accomplished by using processes instead of the current thread implementation.

More delay probing experiments should be performed on the wireless network which has shown unpredictable behavior and high packet drop. It would be interesting also to consider the aspects of delay in mobile CEs (computational elements) connected via a wireless ad-hoc network since nowadays several mobile applications requiring load-balancing are present (e.g. wireless sensors, moving robots performing a joint task).

Finally, further investigations of the newly proposed adaptive and dynamic load-balancing policy (Chapter 6) are needed; more experiments should be conducted on a larger number of nodes to test its performance and more importantly its scalability. Moreover, the update methods of the adaptive parameters C and r_{ij} should be enhanced by observing the impact of the gain values α and β on the stability of the system.

References

[1] Cygwin website. http://www.cygwin.com.

[2] Planetlab website. http://www.planet-lab.org.

[3] RIPE NCC test traffic measurements. http://www.ripe.net/test-traffic.

[4] RIPE NCC website. http://www.ripe.net.

[5] C. T. Abdallah, N. Alluri, J. D. Birdwell, J. Chiasson, V.Chupryna, Z. Tang, , and T. Wang. A linear time delay model for studying load balancing instabilities in parallel computations. In *The International Journal of System Science*, 2003.

[6] D.A. Bader, B.M.E. Moret, and L. Vawter. Industrial applications of high-performance computing for phylogeny reconstruction. SPIE ITCom2001, August 2001.

[7] S. A. Banawan and J. Zahorjan. Load sharing in heterogeneous queueing systems. In *IEEE Infocom '89*, pages 731–739, April 1989.

[8] Sayed A. Banawan and Nidal M. Zeidat. A comparative study of load sharing in heterogeneous multicomputer systems. In *25th Annual Proceedings of the Simulation Symposium*, pages 22–31, April 1992.

[9] Wolfgang Becker. Dynamic balancing complex workload in workstation networks - challenge, concepts and experience. In *Proceedings High Performance Computing and Networking (HPCN) Europe*, 1995. http://www.informatik.uni-stuttgart.de/ipvr/as/projekte/hicon/HPCNPaper.doc.html.

[10] J. Douglas Birdwell, John Chiasson, Zhong Tang, Chaouki Abdallah, Majeed M. Hayat, and Tsewei Wang. Dynamic time delay models for load balancing part

I: Deterministic models. *CNRS-NSF Workshop: Advances in Control of Time-Delay Systems*, January 2003.

[11] C.J. Bovy, H.T. Mertogimedjo, G. Hooghiemstra, H. Uijterwaal, and P. Van Mieghem. Analysis of end-to-end delay measurements in the internet. In *Proceedings of the Passive and Active Measurements Workshop (PAM2001), Ft.Collins*, March 2001.

[12] Thomas L. Casavant and Jon. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. In *IEEE Transactions on Software Engineering*, volume 14:2, pages 141 – 154, February 1988.

[13] J. Chiasson, J. D. Birdwell, Z. Tang, and C.T. Abdallah. The effect of time delays in the stability of load balancing algorithms for parallel computations. IEEE CDC, Maui, Hawaii, 2003.

[14] J. Chiasson, Z. Tang, J. Ghanem, C. T. Abdallah, J. D. Birdwell, M. M. Hayat, and H. Jerez. The effect of time delays on the stability of load balancing algorithms for parallel computations. In *IEEE Transactions on Control Systems Technology, Submitted.*

[15] Yuan-Chieh Chow and Walter H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transactions on Computers*, volume C-28(5):pages 354–361, May 1979.

[16] Douglas E. Comer and David L. Stevens. *Client-Server Programming and Applications, BSD Socket Version with ANSI C*, volume 3 of *Internetworking with TCP/IP*. Prentice Hall, second edition, 1996.

[17] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, volume 7(2):pages 279–301, October 1989.

[18] S. Dhakal, M. M. Hayat, J. Ghanem, C. T. Abdallah, H. Jerez, J. Chiasson, and J. D. Birdwell. *Advances in Communication Control Networks, in the series Lecture Notes in Control an Information Sciences (LCNCIS)*, chapter On the optimization of load balancing in distributed networks in the presence of delay. Springer-Verlag: Berlin, 2004.

[19] S. Dhakal, B.S. Paskaleva, M. M. Hayat, E. Schamiloglu, and C. T. Abdallah. Dynamical discrete-time load balancing in distributed systems in the presence of time delays. IEEE CDC, Maui, Hawaii, 2003.

[20] Sagar Dhakal. Load balancing in delay-limited distributed systems. Master's thesis, Department of Electrical and Computer Engineering, The University of New Mexico, December 2003.

[21] Fotis Georgatos, Florian Gruber, Daniel Karrenberg, Mark Santcroos, Ana Susanj, Henk Uijterwaal, and Rene Wilhelm. Providing active measurements as a regular service for isps. In *Proceedings of the Passive and Active Measurements Workshop (PAM2001), Amsterdam*, April 2001.

[22] J. Ghanem, C. T. Abdallah, M. M. Hayat, S. Dhakal, J.D Birdwell, J. Chiasson, and Z. Tang. Implementation of load balancing algorithms over a local area network and the internet. 43rd IEEE Conference on Decision and Control, Submitted, Bahamas, 2004.

[23] J. Ghanem, S. Dhakal, C. T. Abdallah, M. M. Hayat, and H. Jerez. On load balancing in distributed systems with large time delays: Theory and experiments. IEEE Mediterranean conference on control and automation, Turkey, 2004.

[24] Majeed M. Hayat, Sagar Dhakal, Chaouki T. Abdallah, J. Douglas Birdwell, and John Chiasson. Dynamic time delay models for load balancing part II: A stochastic analysis of the effect of delay uncertainty. *CNRS-NSF Workshop: Advances in Control of Time-Delay Systems*, January 2003.

[25] Gerard Hooghiemstra and Piet Van Mieghem. Delay distributions on fixed internet paths. Technical Report 20011031, Delf University of Technology, 2001.

[26] Information Sciences Institute. Transmission control protocol darpa internet program protocol specification. RFC 793, September 1981. http://www.ietf.org/rfc/rfc793.txt.

[27] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM '88*, August 1988.

[28] Karim Y. Kabalan, Waleed W. Smari, and Jacques Y. Hakimian. Adaptive load sharing in heterogeneous systems: Policies, modifications, and simulation. *International Journal of Simulation, Systems, Science and Technology*, volume 3:pages 89–100, June 2002.

[29] S.N.V. Kalluri, J. JàJà, D.A. Bader, Z. Zhang, J.R.G. Townshend, and H. Fallah-Adl. High performance computing algorithms for land cover dynamics using remote sensing data. In *International Journal of Remote Sensing*, volume 21:6, pages 1513–1536, 2000.

[30] Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, SE-13(1), January 1987.

[31] Peter Xiaoping Liu, Max Q.-H. Meng, Xiufen Ye, and Jason Gu. End-to-end delay boundary prediction using maximum entropy principle (mep) for internet-based teleoperation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2701–2706, 2002.

[32] Hazem Hamed Lopa Roychoudhuri, Ehab Al-Shaer and Greg Brewster. On studying the impact of the internet delays on audio transmission. IEEE Workshop on IP Operations and Management (IPOM'02), 2002.

[33] William Osser. Automatic process selection for load balancing. Master's thesis, University of California at Santa Cruz, June 1992.

[34] J. Postel. Internet protocol. RFC 791, September 1981. http://www.ietf.org/rfc/rfc0791.txt.

[35] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* The Art of Science Computing. Cambridge University Press, second edition, 1992.

[36] Vikram A. Saletore. A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks. In *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 994–999, Charleston, SC, April 1990.

[37] R. Sandoval-Rodriguez, C.T. Abdallah, and P.F. Hokayem. Internet-like protocols for the control and coordination of multiple agents with time delay. Houston, TX, 2003. IEEE International Symposium on Intelligent Control.

[38] R. Sandoval-Rodriguez, C.T. Abdallah, P.F. Hokayem, and E. Schamiloglu. Robust mobile robotic formation control using internet-like protocols. IEEE Conference on Decision and Control, December 2003.

[39] S. Shenker and A. Weinrib. Asymptotic analysis of large heterogeneous queuing systems. In *Proceedings of the 1988 ACM SIGMET-RICS Conference*, pages 56–62, May 1988.

[40] Scott Shenker and Abel Weinrib. The optimal control of heterogeneous queueing systems: A paradigm for load-sharing and routing. *IEEE Transactions on Computers*, volume 38(12):pages 1724–35, December 1989.

[41] K. G. Shin and Y. C. Chang. Load sharing in distributed real-time systems with state-change broadcasts. *IEEE Transactions on Computers*, volume 38(9):pages 1124–1142, September 1989.

[42] W. Richard Stevens. *Networking APIs: Sockets and XTI*, volume 1 of *UNIX Network Programming.* Prentice Hall, second edition, 1998.

[43] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2002.

[44] Andras Veres and Miklos Boda. The chaotic nature of TCP congestion control. In *Proceedings of the IEEE Infocom*, pages 1715–1723, 2000.

[45] Abel Weinrib and Scott Shenker. Greed is not enough: Adaptive load sharing in large heterogeneous systems. In *Proceedings of the IEEE Infocom '88*, pages 986–994, March 1988.

[46] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, volume 4:pages 979–993, September 1993.