Order Number 9409710

Task allocation and redistribution in distributed real-time systems

Hou, Chao-Ju, Ph.D.

The University of Michigan, 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

TASK ALLOCATION AND REDISTRIBUTION IN DISTRIBUTED REAL-TIME SYSTEMS

by

Chao-Ju Hou

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering: Systems)
in The University of Michigan
1993

Doctoral Committee:

Professor Kang G. Shin, Chair
Professor John F. Meyer
Professor Stephen Pollock
Assistant Professor Chinya V. Ravishankar
Associate Professor Demosthenis Teneketzis

To my parents

ACKNOWLEDGEMENTS

My deepest gratitude goes to my advisor Professor Kang G. Shin for his constant guidance, encouragement, and support throughout the course of this work. He has always encouraged me to cultivate my potential when I was not even sure I was capable of pursuing a Ph.D. During the course of my study, he has always been eager to discuss our problems and has made many inspiring comments and insightful suggestions without which this work would not have been commpleted. Moreover, in spite of the tremendous demand on his time, he has always been concerned my academic progress and personal life. I would also like to express my appreciation to Professors John Meyer, Demosthenis Teneketzis, Stephen Pollock, and Chinya Ravishankar for serving on this doctoral committee, and for their constructive suggestions on this dissertation.

I have greatly benefited from my discussions with several previous and present members of the Real–Time Computing Laboratory. In particular, Yi–Chieh Chang introduced me the topic of load sharing and indicated to me possible directions to pursue. The early work on real–time task allocation and scheduling by Dar-Tzeng Peng also inspired the work constituting Chapter 2 of the thesis. Alan Olson helped me understand the Condor package based on which we implemented our load sharing mechanism. Thomas Kaeppel Tsukada collaborated with me on the implementation and experiments of our load sharing mechanism.

I would also like to gratefully acknowledge the Department of EECS, the Office of Naval Research, and National Science Foundation for providing financial support during the course of my graduate study. Thanks to James Dolter and Daniel Kiskis for helping me with many LaTeX problems encountered in preparing this document. Thanks to Atri Indiresan for many discussions on coursework and on student social life. Thanks also to Wafa Wei, Ya-Wen Ko, Shou-Te Chang, Petra Deker, and Anant Chiarawongse for making my stay in Ann Arbor more enjoyable.

A special thank goes to Muh-Ling Ger for his help, encouragement, and understanding, and for our very special friendship. Finally, I deeply thank my parents for their weekly letters that went across the Pacific Ocean and were never absent for a single week during the last six years, for their constant support and never–ending love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure

LIST OF APPENDICES

Appendix

CHAPTER 1

INTRODUCTION

1.1 Background

The availability of inexpensive, high-performance processors and high-capacity memory chips has made it attractive to use distributed computing systems for real-time applications. These systems offer several advantages such as parallel computation, performance scaling, and graceful degradation in case of component failures. Moreover, a distributed system is ideally suited for environments with considerable physical separation among the components to be controlled. However, these potentially attractive features cannot be realized without careful coordination of processing nodes (PNs) and judicious distribution/redistribution of application tasks in the system. The main goal of this dissertation is to address the problems of statically allocating modules of periodic tasks and dynamically redistributing non–periodic tasks to PNs in a distributed real–time system.

In a real–time system, the value of a computation depends not only on the logical correctness of the results, but also on the time at which the results are produced [SR88]. Hence, time is the most important resource in the system, and each real–time task is characterized by a *task deadline*. A task that is not completed within its deadline after its invocation/release is considered failed, regardless of whether it is eventually completed or not. Consequently, the performance of a real–time system is assessed on a *per-task* basis, and the primary performance objective is no longer to minimize *average task response time*, but rather, to minimize the probability of a task failing to be correctly completed by its deadline, which is termed the *probability of dynamic failure*, P_{dyn}, in [KS83, SKL85].

Real-time tasks are either periodic or non-periodic. A periodic task is invoked at fixed time intervals and constitutes the base load of the system. Its attributes, such as the required resources, the execution time, and the invocation period, are usually known *a priori*. A non-periodic task, on the other hand, is invoked randomly in response to environmental stimuli, especially to unanticipated abnormal situations. To ensure that the

execution of both periodic and non–periodic tasks must be both logically correct and completed before their deadline, a real–time task system is usually handled with the following phases:

Task decomposition: partitions periodic tasks into a set of communicating modules characterized by their required execution times, release times, latest completion times, and precedence relations.

Module allocation: statically allocates modules of periodic tasks to PNs in a distributed real–time system subject to task precedence and timing constraints.

Load redistribution/sharing: dynamically distributes non–periodic tasks as they arrive according to the load state and task attributes of each PN.

Local scheduling: uses the minimum–laxity–first–served (MLFS) discipline to schedule all tasks/modules distributed/allocated to a node, because the MLFS discipline has been shown in [HTT89] to, on average, outperform others in reducing P_{dyn}.

Partitioning tasks is usually based on some application-dependent criterion and the system architecture under consideration, which is not the intent of this dissertation; see [PS87] for an example of partitioning real-time tasks into modules/activities. In this dissertation, we consider instead the issues of allocating periodic task modules to ensure their timely completion, and redistributing non–periodic tasks as they arrive at "incapable" nodes which do not have enough resources to complete them in time.

Allocation of Periodic Task Modules: Since the attributes of periodic tasks are usually known *a priori*, periodic tasks are decomposed into a set of communicating modules, and are represented by a *task flow graph* (TFG). To fully specify task behaviors, task invocations within a specific period during which task behaviors will repeat for the entire mission are considered. Such a period is called the *planning cycle* of periodic tasks and will be elaborated on in Chapter 2. The problem is then to allocate all the modules in a planning cycle to PNs in the system with respect to their precedence and timing constraints to ensure that all periodic tasks meet their timing requirements.

The problem of allocating tasks/modules in a distributed system has been studied by many researchers with respect to different objective functions subject to different constraints. These objective functions can be roughly grouped into four categories:

O1. Minimization of total computation and communication times in the system [MLT82, Sto77, Lo88, Hou90, WM93]. In the case of homogeneous systems, this objective function reduces to the minimization of the total interprocessor communication time.

O2. Load balancing by minimizing the statistical variance of processor utilization [BT83, TT85] or by maximizing the total rewards in the semi-Markov process with rewards that models the system [CA82].

O3. Minimization of maximum computation and communication times on a PN, the objective function of which was termed the maximum turnaround time in [ST85], the bottleneck processor time in [CS87, CL87], and the system hazard in [PS89].

O4. Maximization of the reliability function of both PNs and communication links [SW89, SWG92].

Different objective functions lead to different optimality conditions and different allocation results. The first two objectives are suitable for a distributed system executing multiple simultaneous non real-time applications, where maximizing the total throughput or minimizing the *average* response time is the main concern. However, for real-time systems, the logical and timing correctness of each individual task must be considered, because failure to correctly complete a task in time could cause disastrous consequences. Thus, the third objective function which is based on the worst-case behavior is more suitable for assessing the timeliness of real-time systems, while the fourth objective function that incorporates reliability into task/module allocation is more suitable for assessing logical correctness.

The allocation problem has also been shown to be NP–hard for most existing problem formulations [GJ79, FB89], and some form of enumerative optimization and/or local search approaches must be sought. As was reviewed in [MLT82, Lo88, BT83, CA82, ST85], most existing methods are based on graph theory [Sto77, Lo88], mathematical programming/enumeration [MLT82, ST85, PS89], heuristics/approximations [BT83, CS87, CL87], or Markov decision theory [CA82].

In this dissertation, we address the problem of allocating periodic task modules in a planning cycle subject to their task precedence and timing constraints to maximize the probability of completing each task with *both* logical and timing correctness, $P_{ND} \triangleq 1-P_{dyn}$. By "allocation," we mean the *assignment* of modules coupled with the *scheduling* of all modules assigned to each PN. Using the branch–and–bound (BB) technique, a module allocation (MA) algorithm is proposed to find an "optimal" allocation that minimizes the probability of dynamic failure, P_{dyn}.

Load Redistribution/Sharing: Since non–periodic task arrivals might be temporarily uneven among PNs in a distributed system, the allocation of periodic task modules might be "load–unbalanced" among the nodes, and/or the processing power might vary from node

to node, some nodes may get temporarily overloaded while others are left underloaded/idle [KK93]. Livny and Melman [LM82] showed that even in a network of autonomous nodes, with a large probability at least one node is idle while many tasks are being queued at other nodes. Thus, we need an effective load redistributing or *load sharing* (LS) method to enable "capable" nodes to share the loads of "incapable" ones and to maximize the probability of non–periodic tasks meeting their deadlines. By "capable node," we mean a node with enough resources available to complete transferred–in task(s) in time.

LS in a distributed real–time system is different from that in a general-purpose system in that the latter tries to either achieve perfect load balancing among the nodes and/or minimize average task response time, whereas the former is intended to minimize P_{dyn}. Upon arrival of a real–time task, each node determines whether or not it can complete this task in time. If it can, the node will execute the task locally; otherwise, some other capable node will be chosen to execute the task. As was discussed in [ELZ86, SKS92], LS in a distributed system is dictated by three component policies: the *transfer policy* for determining *when* to transfer a task, the *location policy* for determining *where* to transfer the task, and the *information policy* for determining *how* each node collects state information from other nodes. In the context of real–time applications, the transfer policy determines whether or not a task can be completed in time locally, and the location policy determines which other node is most likely to complete the task to be transferred in time.

According to the properties of the three component policies, LS schemes can be classified into three categories: deterministic, probabilistic and dynamic/adaptive [NH85, YL84, HL86]. A deterministic approach allows an overloaded/incapable node to transfer "overflow" tasks — tasks that cannot be completed locally in time — with a fixed pattern, e.g., all overflow tasks on node i are transferred to node j. A probabilistic approach, on the other hand, transfers tasks with pre-specified probabilities, e.g., an overloaded/incapable node i will transfer its overflow tasks to node j with probability P_{ij}.

By contrast, an adaptive approach uses state information for their LS policies. The state of a node may be the number (or queue length, QL), or the cumulative task execution time (CET), of tasks queued for execution on the node, the number and type of available resources, or a function or combination thereof. The node makes LS decisions based on the information collected via either periodic state broadcasts [Sta84, BS85, HL86, Sta85], or state probing or bidding [Smi80, LM82, KF84, WM85, SRC85, NXG85, CL86, CK87, KC87, Zho88, WS88, RSZ89, MTS89b, MTS89a, OK92], or state–change broadcasts [LM82, HJ87, SC89a, SC89b, SH91, SC90]. Both deterministic and probabilistic approaches do not use state information, and thus, cannot react to dynamic situations. Because an adaptive

approach can adjust itself to dynamically–changing conditions, it is naturally expected to outperform non-adaptive approaches in meeting task deadlines.

Several issues have to be considered in designing an adaptive LS mechanism for real–time applications. The requirement that each real–time task should meet its deadline suggests that the transfer policy not be of the *static* threshold type commonly used for general–purpose systems [ELZ86, MTS89b], but should instead depend on the *laxity* of each task — the latest time the task must start execution in order to meet its deadline. In other words, upon arrival of a task, a node determines whether or not it can complete the task in time by checking the CET on the node in front of the newly arrived task is less than or equal to the laxity of the arrived task.

A node with an overflow task uses the location policy and the state information gathered to locate a candidate node for task transfer by checking whether or not the candidate node has the surplus resource necessary to complete the task in time. Two issues need to be considered when choosing the receiver of an overflow task:

(a) the probability of transferring the overflow task to an 'incapable node' must be minimized.

(b) excessive task transfers resulting from task collisions must be avoided. A *task collision* is said to occur if the "guarantee" of one or more tasks queued at a node is deprived due to the arrival of a new tighter–laxity task. (By "guarantee", we mean the node has enough resources to complete the task of interest in time *upon its arrival*. An existing guarantee may be deprived later because of tighter–laxity task arrivals under the MLFS policy.)

The choice of information policy has an impact on the performance as well. LS based on the periodic exchange of state information requires a good or optimal means of determining the period of information exchange, since the accuracy of state information when a LS decision has to be made depends heavily on this period. On the other hand, LS based on bidding/state probing generates at least two additional messages per bidding/probing, introducing time and communication overheads, and may thus be detrimental to the timely completion of real–time tasks. Moreover, the resulting LS performance is very sensitive to the variation of communication delay. LS that requires to update the state information only in case of state–region changes has the advantage of maintaining more up–to–date state information and collecting it inexpensively before it is needed for a LS decision. However, the performance is susceptible to node failures, because the node failure is usually detected through communication timeouts, and for LS with state–region–change broadcasts, if a

node has been silent (i.e., does not broadcast its state–region changes) for a long time, other nodes have no way of knowing whether this is an indication of the node's failure or a coincidence of task arrival and completion/transfer activities alternating on the node (so that the node's state remains in the same state region).

In contrast to other LS approaches proposed for general–purpose distributed systems, we carefully tailor the transfer policy, the information policy, and the location policy to handle real–time applications, to solve potential problems in a distributed real–time environment, and to ensure each non–periodic task to be completed before its deadline on a *best effort* basis.

1.2 Research Objectives

Although distributed systems offer many advantages (as discussed in Section 1.1), they also present several challenging problems. Our research addresses one of them: task/module allocation and load sharing in distributed real–time systems. The primary objectives of our research are to meet the requirement in hard real–time systems that the execution of both periodic and aperiodic tasks must be not only logically correct but also completed before their deadline. Specifically, we address the issue of "optimally" (in the sense to be defined later) allocating periodic task modules to PNs so as to fully utilize the inherent parallelism, modularity, and reliability of the system while alleviating the "saturation effect" [MLT82] caused by excessive interprocessor communication of data and control messages. We also treat the dynamic redistribution of non–periodic tasks as an adaptive load sharing (LS) problem, and design a LS mechanism to reduce the probability of a non–periodic task missing its deadline.

In particular, we have identified several important issues related to MA and LS in distributed real–time systems and have developed solutions to them in subsequent chapters:

Combining assignment with scheduling in MA: Because of the timing aspects embedded in the objective function used in real–time systems, the performance of any resulting module assignment strongly depends on how the assigned modules are scheduled. That is, not only a module allocation method is needed to assign task modules to PNs, but also a module scheduling algorithm should be used to schedule all modules assigned to each PN to ensure that all tasks may be completed in time. Specifically, modules which constitute periodic tasks are characterized by its (1) required execution time, (2) *earliest release time* which is derived from the task invocation times and precedence relations, and (3) *latest completion time* which is derived from task deadline and precedence relations.

The latter two quantities form the timing requirements imposed on periodic tasks and the communication times among modules into the characteristics of modules. Whether or not to assign a module to a PN depends on whether or not the PN under consideration has enough resources available to complete the module between its release time and its latest completion time.

Characterization of state inconsistency: No matter which information policy is used to collect state information, the state information gathered may be out–of–date due to the communication delay incurred in state–information collection and task transfer [CK87]. That is, a node's *observed* states of other nodes may be different from their *true* states at the time of making LS decisions. This inconsistency often causes a node to transfer an overflow task to an actually incapable node, and degrades the performance of adaptive LS (as was analyzed in [MTS89b, MTS89a]). An on–line characterization approach should be used to capture the inconsistency between a node's "observed" state and the corresponding true state with prior/posterior distributions. Each node should, instead of hastily believing what it observed, estimates the *true* states of other nodes with these distributions and *observed* states collected via the information policy, and makes LS decisions based on the *estimated* states to reduce the probability of transferring a task to an incapable node.

Consideration of future tighter–laxity task arrivals: Under the MLFS scheduling discipline, a node determines whether or not to queue a task with laxity ℓ upon its arrival by checking if the CET contributed by tasks with laxity $\leq \ell$ at the node is less than or equal to ℓ. If a node cannot complete a newly-arrived task in time or the deadline of one or more tasks in its queue is to be missed as a result of inserting the newly–arrived task into its schedule, the node has to determine — based on some state information — candidate receiver(s) for task transfer(s). In case of heterogeneous task arrivals at each node (e.g., different task arrival rates, distributions of task laxity, or distributions of task execution time), transferring an overflow task $\mathcal{T}$ of laxity ℓ to the node with the least CET may not be a good choice if that node happens to have a large composite task arrival rate or most tasks arrived at that node happen to have tighter laxities than the transferred–in task. For example, a node may become easily incapable as a result of a high local arrival rate or simultaneous task transfers from multiple nodes to the same node. Transferring $\mathcal{T}$ to such a node may not be a good decision, even if the node was idle/underloaded at the time of locating the receiver of $\mathcal{T}$. Those tasks subsequently arrived at the node may have to be transferred as a result of its acceptance of $\mathcal{T}$. $\mathcal{T}$ may have to be transferred out again due

to the subsequent arrival of a tighter–laxity task under the MLFS scheduling discipline. That is, tighter–laxity tasks arrived after the arrival of $\mathcal{T}$ but prior to its execution may deprive $\mathcal{T}$ of its "guarantee". Task collisions may thus occur and excessive task transfers may ensue. Consequently, the impact of future task arrivals should be taken into account in making a LS decision in real–time systems.

Tuning of best timeout period for LS with aperiodic broadcasts: As discussed in Section 1.1, failure of a node is usually diagnosed by the other nodes through communication timeouts. A node is diagnosed as faulty if it has not communicated with other nodes for a timeout period. However, for LS with aperiodic state–region change broadcasts, the communication pattern of a node dynamically changes with system load, the attributes of tasks arrived at the node, and the state the node was initially in. Hence, the best timeout period used to diagnose whether a node is faulty or not should also be on–line adjusted with these parameters. A timeout mechanism with an on–line adjustable timeout period should be incorporated into LS with aperiodic state–region change broadcasts to avoid sending overflow tasks to failed nodes. In particular, a best timeout period should be determined on–line to maximize the probability of detecting node failure while keeping the probability of incorrect diagnosis below a pre–specified level of tolerance.

1.3 Approaches

Design of MA algorithm and LS mechanism: The goal of this dissertation is to formulate and solve problems related to task management, or more specifically, task/module allocation and load redistribution, in distributed real–time systems. We formulate problems outlined in Section 1.2 in a well–defined analytic framework, and employ the branch–and–bound method, Bayesian decision theory, queueing theory, hypothesis testing, Markov modeling, and randomization technique to develop solutions in a general setting, which would be applicable to a variety of real–time systems.

We would also like to model, validate (via event–driven simulations), apply our solutions to a specific experimental distributed system, and implement our solutions so as to demonstrate their effectiveness and develop a base for experimenting with real applications. Specifically, we have carried out the following additional tasks:

Analytic modeling of LS schemes using semi–Markov models: We develop analytic semi-Markov models to comparatively evaluate the proposed LS mechanism as well as three other schemes: no LS, LS with random selection of a receiver node, and

LS with perfect information. Specifically, we model the evolution of a node's load state as a continuous–time semi–Markov process, where CET is used to describe the workload of a node. We not only address the fundamental differences among different LS schemes to model their design diversity, but also take into account of implementation overheads to build a more practical model for accurately analyzing the tradeoff between the design complexity and the resulting benefit. Several metrics relevant to real–time performance are derived from these models. We then compare the proposed LS mechanism against other LS schemes using these performance metrics. We also validate the analytic models with event–driven simulations.

Application of integrated LS to HARTS: No matter which information or location policy is used, we must consider an underlying communication subsystem that supports all LS–related communications, i.e., exchange of state information and task transfers. In particular, the interconnection network and protocols affect how tasks or broadcast messages are routed and whether or not tasks/messages can be delivered by a certain time [SH90], and the underlying switching scheme determines whether tasks/messages may be queued at intermediate nodes or not. That is, the underlying communication subsystem should be taken as an integrated part of the LS mechanism under consideration as far as the performance is concerned.

We conduct an integrated LS study on HARTS (Hexagonal Architecture for Real-Time Systems) [Shi91], an experimental distributed real-time system being developed in the Real–Time Computing Laboratory at Michigan. Specifically, we use the proposed LS mechanism to coordinate the nodes in HARTS (which are interconnected by a C–wrapped hexagonal mesh [CSK90]) to evenly share overflow tasks. We also use the HARTS routing and broadcasting algorithms in [CSK90, KS91b] for transferring tasks and broadcasting state changes, and the virtual cut-through switching scheme [KK79] implemented in HARTS [DRS91] for inter-node communication. By exploiting/integrating features of these communication–related algorithms for/into LS, we rigorously analyze the integrated performance of LS in HARTS.

The results obtained from the analytic models are validated through event-driven simulations, and used to study the impact of varying various design parameters on the performance of LS while considering the details of LS–related communication activities.

Prototyping the LS mechanism with software: We have implemented a preliminary version of the proposed LS mechanism based on the Condor software package developed by

researchers at the University of Wisconsin [LLM88]. Both the transfer and location policies in Condor are realized by a centralized component named the *central manager* node. This centralized component makes the LS performance susceptible to single–component failures. Besides, the periodic information policy used in Condor introduces a potential bottleneck of network traffic at the central manager, and makes the fine-tuning of a reporting period a crucial performance issue. We demonstrate how to enhance the fault tolerance capability and the performance of Condor by configuring the functions of the central manager into participating nodes using the schemes.

1.4 Outline of the Dissertation

This dissertation is organized as follows. In Chapter 2, we design a module allocation algorithm (**MAA**) to allocate periodic task modules to PNs and then subsequently schedule all modules assigned to each PN with respect to their precedence and timing requirements so that all periodic tasks may be completed in time. We first model the task system within a planning cycle with a task flow graph to describe computation and communication modules as well as the precedence constraints among them. To incorporate both timing and logical correctness into module allocation, we use the probability of no dynamic failure, P_{ND}, as the objective function. The **MAA** is then applied to find an optimal allocation of task modules. To reduce the computational complexity, we derive (1) a dominance relation from the requirement of timely completion of tasks and used it to avoid generating vertices in the state-space search tree which never lead to an optimal solution, and (2) an upper bound of P_{ND} for every partial allocation with which the **MAA** uses to prune the intermediate vertices in the search tree. We also perform extensive numerical experiments to evaluate the effectiveness and practicality of the **MAA**.

In Chapter 3, we design an effective LS mechanism by tailoring LS component policies to enable capable nodes to share the loads of overloaded ones so that the probability of dynamic failure, P_{dyn}, is minimized. An important feature of the design is the use of (1) time–stamped region–change broadcasts and prior/posterior probability distribution of load state to characterize the inconsistency between the state a node observes and the true systemwide state and (2) Bayesian decision theory to estimate the true states of other nodes. By characterizing the inconsistency between a node's 'observed' state and the corresponding true state with prior/posterior distributions, and by using Bayesian decision theory for LS decisions, the node can first estimate the states of other nodes, and then use them to reduce the probability of transferring a task to an incapable node. We study (via event–driven

simulation) the performance of the LS mechanism, and investigate the effect of statistical fluctuations of task arrivals on the LS performance.

Chapter 4 presents the analytic models which assess the performance of our proposed LS mechanism as well as other existing LS schemes. The evolution of a node's load state is modeled as an $M^{[k]}/D/1$ queue with bulk arrivals, where the cumulative execution time (CET) is used to describe the workload of a node. Both fundamental differences among different LS schemes and computation/communication overheads associated with their implementation are taken into account in the models. Several metrics relevant to real–time performance are then derived from these models with which we use to evaluate the proposed LS mechanism against other LS schemes.

In Chapter 5, we take into account future task arrivals in the LS decision to minimize not only the probability of transferring an overflow task to an incapable node, but also the probability that a remote candidate node fails to complete the transferred task in time because of future arrivals of tight–laxity tasks there. The probability that the "guarantee" of a transferred task will be deprived (after its transfer) by future tighter–laxity tasks on a remote node is approximated using queueing analysis, and is used as an index of the likelihood of future tighter-laxity task arrivals and its impact on the node's capability of completing the transferred task. All parameters needed for calculating the probability of interest (and thus for making the LS decision) are on–line collected/estimated with Bayesian estimation technique. We examine the performance improvement resulted from the consideration of future task arrivals in minimizing the probability of dynamic failure, task collisions, and excessive task transfers.

Chapter 6 addresses the problem of incorporating a timeout mechanism with on–line adjustable timeout periods into LS with state-region change broadcasts. We formulate the problem of determining the 'best' timeout period as a hypothesis testing (HT) problem, and maximize the probability of detecting node failures subject to a pre-specified probability of falsely diagnosing a healthy node as faulty. All the task parameters needed in HT are again collected on–line with Bayesian estimation. We study the performance improvement of the LS mechanism made by combining the on–line parameter estimation, the timeout mechanism, and a few extra, timely broadcasts in reducing the probability of dynamic failure.

Chapter 7 presents a rigorous analysis of LS in HARTS that includes all LS–related communication activities, message routing and broadcasting. We adapt our LS mechanism to HARTS by (1) exploiting the topological properties of HARTS for coordinating nodes in sharing loads, (2) employing the routing and broadcasting schemes developed for HARTS

as the facilities for routing (transferred) tasks and broadcasting state–change messages, (3) using the virtual cut–through switching scheme currently implemented in HARTS for message passing. We also construct a continuous–time Markov chain model to describe task arrival and transfer/completion activities on a node under the LS mechanism and a queueing network model to describe the communications activities introduced by LS in HARTS. Then, we integrate these two models to characterize the system operations of HARTS, and derive several measures related to real–time performance.

In Chapter 8, we describe how to implement, based on the Condor software package, an initial version of the proposed LS mechanism which incorporates the ideas presented in Chapter 3. We first give an overview of how Condor daemons collaborate to manage the job queue at each node and locates target nodes. Then, we discuss how to reconfigure Condor daemons and incorporate the proposed LS mechanism into them. We also highlight the implementation features adopted in the proposed mechanism, and discuss related work on alternative design and implementation approaches used by other process migration mechanisms.

This dissertation concludes with Chapter 9, which reviews the contributions of this dissertation and presents a discussion of future directions for the work presented herein.

CHAPTER 2

ALLOCATION OF PERIODIC TASK MODULES WITH PRECEDENCE AND DEADLINE CONSTRAINTS

2.1 Introduction

In this chapter, we develop a module allocation algorithm (**MAA**) with precedence and deadline constraints using the branch-and-bound (BB) method. The probability of completing each task with *both* logical and timing correctness—which was termed in [SKL85] as the probability of no dynamic failure, P_{ND}—is used as the performance metric for locating the optimal solution in the search space. Specifically, P_{ND} is the product of two component probabilities:

- The probability, P_{ND1}, that all tasks within a planning cycle are completed before their deadline. The planning cycle is the time period within which the task invocation behavior repeats itself throughout the entire mission, and thus completely specifies the entire task system. More on this will be discussed in Section 2.2.
- The probability, P_{ND2}, that all PNs are operational during the execution of task modules assigned to them, and the links between communicating PNs[1] are operational for all intermodule communications over these links under a given allocation.

The use of P_{ND} as the objective function is in sharp contrast to the other module allocation approaches reported in the literature which deal with either average task response time or logical correctness, but not both.

We first model the task system with a task flow graph (TG) which describes computation and communication modules as well as the precedence constraints among them. We then use the BB method to search for an optimal module allocation. The computational

[1]By communicating PNs, we mean a pair of PNs to which two communicating modules are assigned under a given allocation.

complexity is reduced by deriving an upper bound of the objective function, $\hat{P}_{ND}$, with which we determine whether to expand or prune intermediate vertices (corresponding to partial allocations) in the state-space search tree. On the other hand, because of the timing aspects embedded in the objective function, P_{ND}, the performance of any resulting assignment strongly depends on how the assigned tasks/modules are scheduled. Thus, when evaluating an upper-bound (exact) objective function for a partial (complete) allocation, we use a module scheduling algorithm (**MSA**) (with polynomial time complexity) to schedule all the modules assigned to a PN by minimizing the maximum tardiness of modules subject to precedence constraints. The **MAA**, combined with the **MSA**, is guaranteed to find the optimal allocation of modules to PNs subject to precedence and timing constraints. By 'allocation,' we mean the *assignment* of modules coupled with the *scheduling* of all modules assigned to each PN.

Shen and Tsai [ST85] minimized the maximum turnaround time but considered only a single invocation of each task. They did not take into account precedence constraints between tasks. Chu *et al.* [CS87, CL87] chose to minimize the bottleneck processor workload for tasks/modules assignment, but their algorithm/analysis was solely based on *mean* task response times, which eliminates the need to consider the scheduling problem. Peng and Shin [PS89] are the first to include the important timing aspects in the objective function, and combine task scheduling with task assignment. They chose to minimize the *system hazard* which is defined as the maximum normalized (with respect to task period) task flowtime. It is, however, not clear how system hazard is related to the probability of no dynamic failure. The restriction on assigning all modules of the same task to a single PN may not always be desirable.

Ramamritham [RSS90] used a heuristic-directed search technique with tunable design parameters to (1) determine whether or not a group of communicating modules should be assigned to the same PN, and (2) allocate different groups of modules to PNs and schedule them with respect to their latest–start–times and precedence constraints. Compared to this work, we use a finer granularity in modeling the real-time task system. (For example, we include probabilistic branches/loops in task graphs and allow communications between periodic tasks.) Although the author of [RSS90] considered fault tolerance via module replication, the degree of replication is pre-determined in an *ad hoc* way without any rigorous justification. Also, no conclusions were made on whether or not his algorithm always leads to an optimal solution. By contrast, we focus on module allocation subject to precedence and timing constraints, as well as on the minimization of P_{ND} which, as mentioned earlier, takes into account both timeliness and logical correctness. Also, as will be demonstrated in

our simulation study later, the **MAA** finds, in all experiment runs, the best allocation at tractable computational costs for task systems with less than 50 modules and/or distributed systems with less than 40 PNs.

The rest of the chapter is organized as follows. In Section 2.2, we discuss how to model real-time task systems. The planning cycle which completely specifies the behavior of task invocations for the entire mission lifetime is defined, and the task flow graphs (TGs) which model the control flow of the applications and the precedence constraints imposed by intertask communications are discussed. Assumptions on the distributed system are also stated there. In Section 2.3, we provide an overview of our module allocation algorithm. The objective functions, $P_{ND1}(x)$ and $P_{ND2}(x)$ — with which an allocation x is assessed in terms of timeliness and logical correctness — are derived in Sections 2.4 and 2.5. In Section 2.6, we address how to achieve both branching and bounding efficiencies. Section 2.7 presents demonstrative examples. This chapter concludes with Section 2.8.

2.2 Task and System Models

2.2.1 The Task System

As discussed in Chapter 1, periodic tasks are invoked at fixed time intervals and constitute the base load of the system. Their attributes are usually known *a priori*. Consequently, periodic tasks are usually partitioned into a set of communicating modules, and are statically allocated module by module to PNs to a distributed system.

Planning cycle: To analyze the behavior of periodic tasks, we only need to consider the task invocations within a specific period, the task behaviors during which will repeat for the entire mission lifetime. Such a period is called the *planning cycle* of periodic tasks and is defined as the least common multiple (LCM) L of $\{p_i : i = 1, 2, ..., N_T\}$, where p_i is the period of a task T_i and N_T is the total number of periodic tasks in the system. That is, the planning cycle is the time interval $[t_0 + kL,\ t_0 + (k+1)L)$, where t_0 is the mission start time, and k is a nonnegative integer. Fig. 2.1 gives an example of a task system $\{T_1, T_2, T_3\}$ with $p_1 = 3$, $p_2 = 4$, and $p_3 = 6$. All tasks in this example are first invoked at time 0, and a planning cycle is the time interval $[0, 12)$. Note that T_i is invoked L/p_i times within a planning cycle, each time with a deadline d_i, not necessarily equal to p_i.[2]

Attributes and precedence constraints among modules: Each task can be decomposed into smaller units, called *modules*. (See [PS87] for a detailed account of how to

[2]The rate monotonic scheduling algorithm in [LL73] is not applicable due to this and the dependence among periodic tasks.

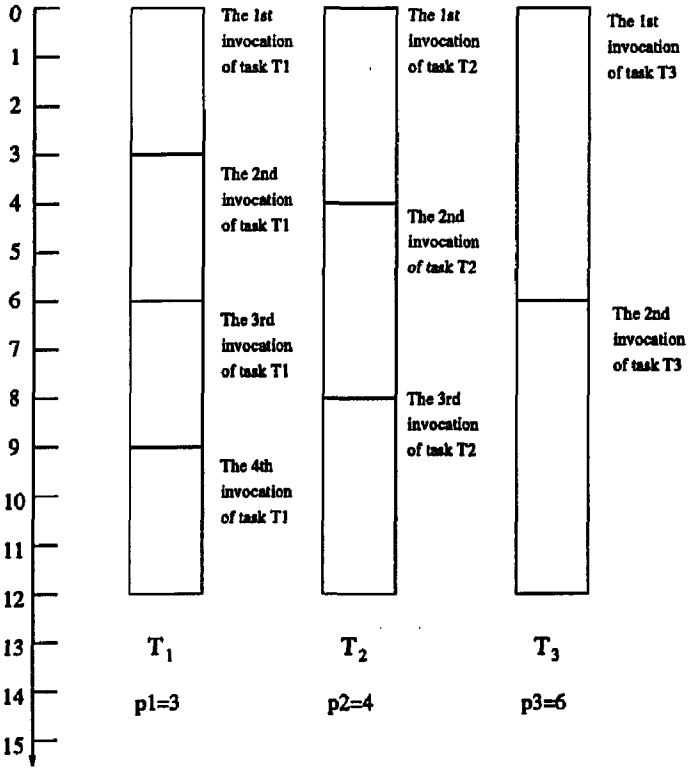


Figure 2.1: The planning cycle which specifies the task system $\{T_1, T_2, T_3\}$ with $p_1 = 3$, $p_2 = 4$, and $p_3 = 6$.

decompose real-time tasks into modules.) Each module M_i requires e_i units of execution time. The execution time of a module could be its worst-case execution time or its real execution time if known. Since extensive simulations and testing are required before putting any real-time system in operation (e.g., fly–by–wire computers), the system designer is assumed to have a good, albeit sometimes incomplete, understanding of either the exact execution time or the worst-case execution time of each module.

The execution order of modules imposes precedence constraints among them. These precedence constraints are of the form $M_i \rightarrow M_j$, meaning that the completion of module M_i of a task enables another module M_j of the same task to be ready for execution (e.g., by letting M_i send a short message to enable M_j's execution and/or update the data variables/files shared between them [CS87, CL87]). On the other hand, tasks communicate with one another to accomplish the overall control mission. The semantics of message communication between two cooperating tasks also impose precedence constraints between the associated modules of these tasks. This kind of precedence constraints is also of the form $M_i \rightarrow M_j$ except that M_i and M_j now belong to different tasks.

If M_i and M_j are assigned to the same PN, communication between them can be achieved via accessing shared memory. Overheads for such communications are usually

much smaller than those when M_i and M_j reside on different PNs. Any two communicating modules that reside on two different PNs will incur interprocessor communication (IPC) which requires extra processing such as packetization and depacketization. IPC introduces a communication delay which is a function of intermodule communication (IMC) volume (measured in data units) and the *nominal link delay* (or delay per data unit) between the two communicating PNs.

It is important to observe that even if exact *module* execution times were known in advance, *task* execution times are not due to, for instance, the existence of probabilistic branches/loops in task flow graphs (to be introduced below) and/or inexact knowledge of IMC/IPC delays.

Task Flow Graph (TG): A TG is commonly used to describe the logical structure of modules, and the communications and precedence constraints among them. A TG is composed of four types of subgraphs: chain, AND-subgraph, OR-subgraph, and loop. A chain is the longest possible sequence of modules connected in series where all but the last module have a single successor. An AND-subgraph consists of more than one branch, all of which must be executed (possibly in parallel) and are enabled by the completion of the module immediately before the AND-subgraph. The module after the AND-subgraph is enabled only after the completion of all branches in the AND-subgraph. A branch of the AND-subgraph may be a single module or any component subgraph of TG. Similarly, an OR-subgraph consists of more than one branch. However, one and only one branch of the OR-subgraph is executed, and the probability of choosing a branch is assumed to be given. Another difference between an AND-subgraph and an OR-subgraph is that a branch of the OR-subgraph could contain no execution object at all. The last type of subgraph, a loop, consists of the loop body with the looping-back probability and the maximum allowable loop count.[3] Like a branch of an AND-subgraph, the loop body may be a single module or any other component subgraph of TG. Fig. 2.2 gives a simple example of a TG.

Communication primitives: The semantics of the most general communication primitive, SEND–RECEIVE–REPLY,[4] can be embedded into precedence relations between modules as shown in Fig. 2.3. If module M_a of task T_i issues a SEND to task T_j, T_i remains blocked, or cannot execute module M_b that follows M_a until the corresponding REPLY from T_j is received. If the module, M_c, responsible for the corresponding communication activity

[3]That is, the number of times a loop can be executed is no more than its maximum loop count. Imposing a maximum loop count for each loop is necessary, since each real-time task must be completed in a finite time.

[4]Other communication primitives, such as QUERY–RESPONSE and WAITFOR [PS87] can always be realized using SEND–RECEIVE–REPLY.

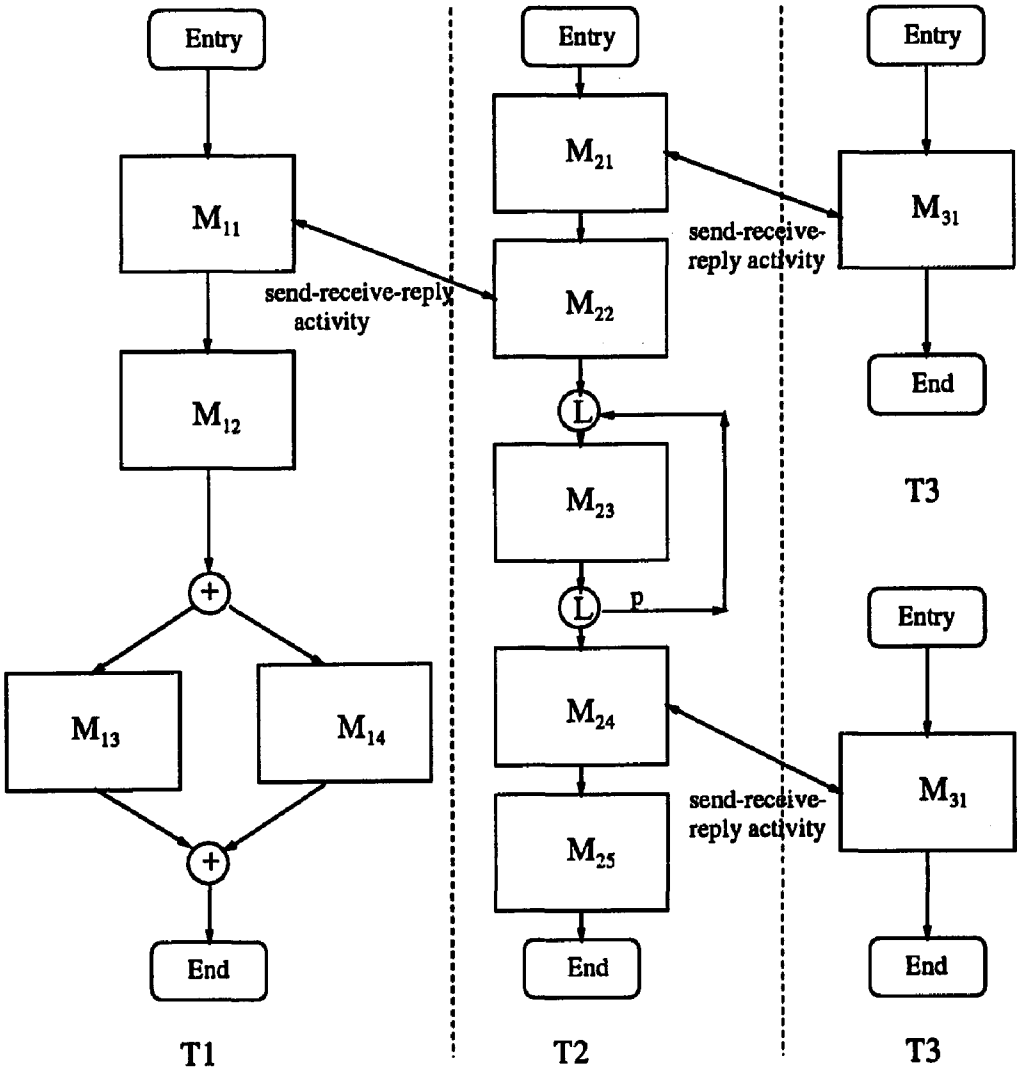


Figure 2.2: An example of task flow graph.

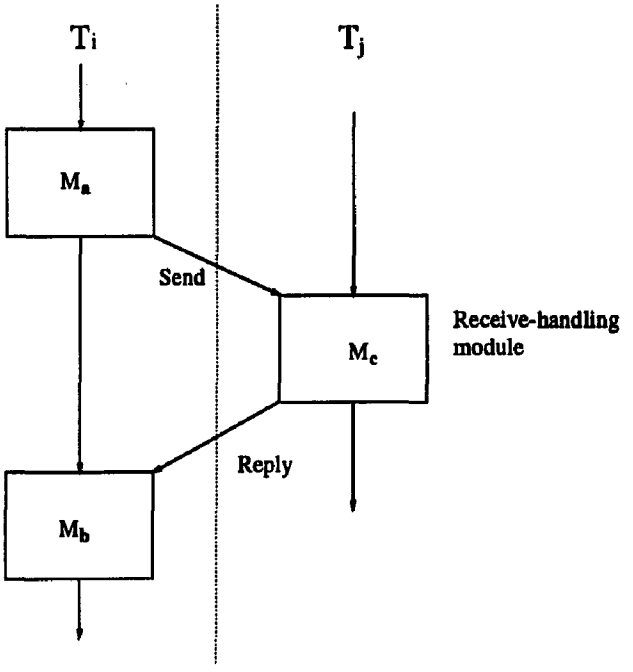


Figure 2.3: Precedence constraints associated with **send–receive–reply**.

on T_j's side executes a RECEIVE before the SEND arrives, T_j also remains blocked. For example, the communication activities between tasks in Fig. 2.2 can be embedded into the precedence constraints between modules as shown in Fig. 2.4.

2.2.2 The Distributed System

The distributed system considered here consists of K_{pn} processing nodes (PNs). For ease of algorithm description, all PNs are assumed to have the same processing power and the same set of resources. (This assumption can be easily relaxed.) The time required by an IMC within a PN is assumed to be negligible, while that between two PNs is expressed as the product of the IMC volume (measured in data units) and the nominal delay (measured in time units per data unit) between the two PNs on which the communicating modules reside.[5] The nominal delay could be the worst-case communication delay experienced by time-constrained messages in the underlying communication subsystem. Here we assume that the communication subsystem and protocol support time-constrained communications (i.e., communication contention is arbitrated with respect to the delivery deadlines of messages), and the worst-case delay experienced by time-constrained messages is bounded and predictable. Two examples of such communication subsystems are the point–

[5]The time for packetization and depacketization is lumped into module execution time for the clarity of algorithm description.

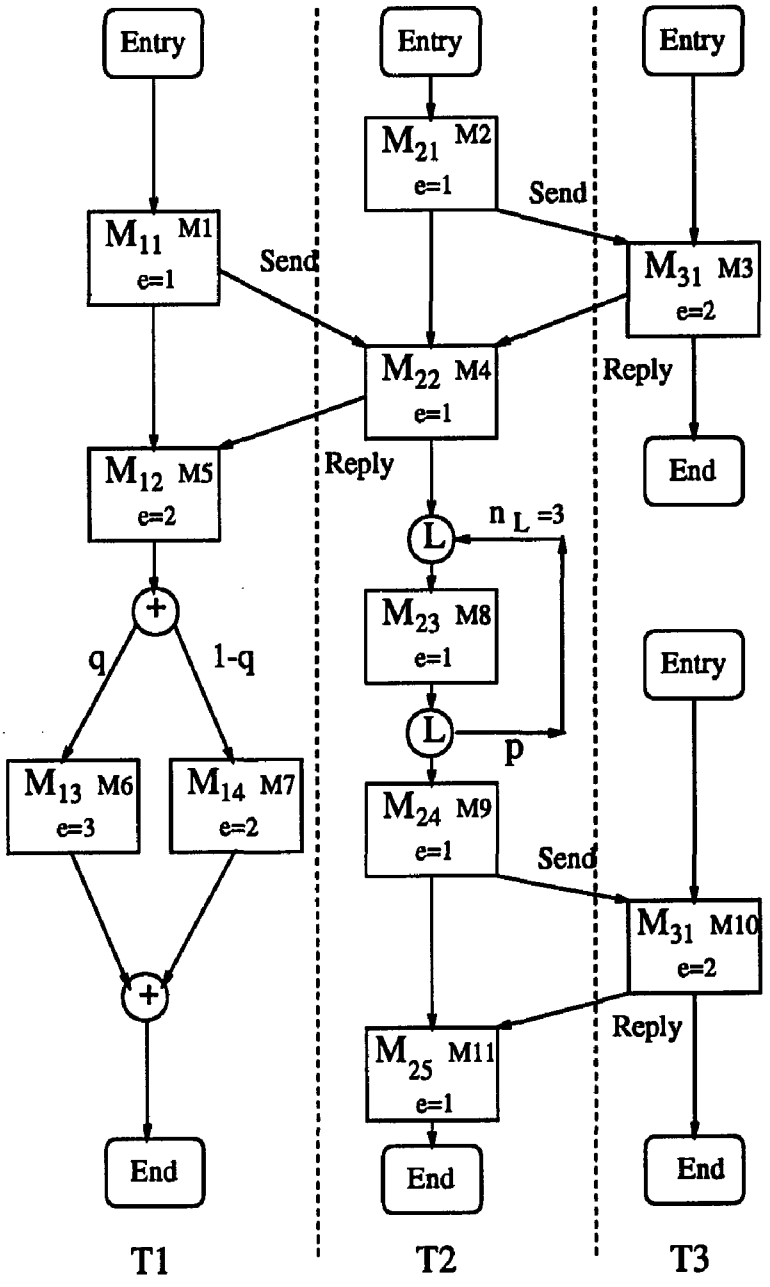


Figure 2.4: The modified TG for the TG in Fig. 2.1 where all communication activities are embedded into precedence relations. The label that appears in the upper right corner of each box is the acyclic number associated with each module.

to–point packet-switched communication subsystem described in [KS91a, ZS92] and the highly responsive token-ring communication subsystem described in [SM89]. No restriction is imposed on the topology of the communication subsystem. Each processing node N_k and each link ℓ_{mn} between N_m and N_n are assumed to fail independently with exponential rates λ_k and $\hat{\lambda}_{mn}$, respectively.

2.3 Module Allocation Algorithm

Let N_M be the number of modules to be allocated within a planning cycle. The module allocation problem can be formulated as that of maximizing $P_{ND}(x) = P_{ND1}(x) \cdot P_{ND2}(x)$ over all possible allocations subject to

$$\sum_{k=1}^{K_{pn}} x_{ik} = 1, \quad \text{for } 1 \leq i \leq N_M,$$

where $x_{ik} = 1$ if and only if M_i is assigned to N_k, $P_{ND1}(x)$ is the probability that all tasks meet their deadlines under allocation x, and $P_{ND2}(x)$ is the probability that all PNs are operational during the execution of modules assigned to them and all communication links are operational during the IPCs that use these links under x. As will be clear later, the precedence constraints among modules are figured in the calculation of module *release times* (to be defined later), and the timing constraints on modules/tasks are considered when evaluating $P_{ND1}(x)$; for example, $P_{ND1}(x) = 0$ if some of the tasks miss their deadline under x. The expressions for $P_{ND1}(x)$ and $P_{ND2}(x)$ will be derived in Sections 2.4 and 2.5 along with the description of the corresponding system parameters.

To solve the above module allocation problem, we use the **MAA** which employs:

Branch–and–bound (BB) method to implicitly enumerate all possible allocations while effectively pruning unnecessary paths in the search tree.

Module scheduling algorithm (MSA) to schedule the modules assigned to each PN subject to precedence constraints and latest module completion times. (The description and analysis of the **MSA** will be given in Section 2.4.1.)

The **BB** method enumerates all possible solutions to a given problem by 'growing' the corresponding search tree. Each intermediate (leaf) vertex in the search tree corresponds to a partial (complete) allocation. This method is composed of two procedures: *branching* and *bounding*. The branching process generates the child vertices of an intermediate vertex x in the search tree, until an optimal solution is completely specified. On the other hand, to avoid generating vertices which will never lead to an optimal solution, we have to calculate

an upper bound[6] of the objective function (UBOF) for each vertex x. Based on the UBOF value of x, one can decide whether or not x may lead to an optimal solution. The rationale behind pruning a vertex based on its UBOF is the bounding process.

In order to derive an optimal solution with as little computation as possible, both branching and bounding processes must effectively prune unnecessary search paths. An effective branching process must minimize the number of child vertices generated for each intermediate vertex x, without eliminating any path to an optimal solution. The rules used by the branching process to limit the growth of the tree are termed *dominance relations*. An effective bounding process must prune as early as possible those vertices that will never lead to an optimal solution. This is achieved by deriving a tight (thus expensive to derive) UBOF for each intermediate vertex x.

The **MAA** which is based on the **BB** method works as follows. The problem of module allocation is represented by a state-space search tree. All modules in the task system are assumed to be numbered in acyclic order such that if $M_i \rightarrow M_j$ then $i < j$. For example, the numbers which appear on the upper right corner of the boxes in Fig. 2.4 give an example of acyclic numbering. The **MAA** begins with a null allocation x_0 which corresponds to the root of the search tree, and allocates modules in the order of their acyclic numbering. Each intermediate (leaf) vertex x in the search tree corresponds to a partial (complete) allocation of modules.

Let $TG(x)$ denote the set of modules which are already allocated under x,[7] and AN the set of active vertices in the search tree to be considered for expansion. AN is determined by the *bounding test*. Expanding a vertex $x \in AN$ corresponds to allocating the module, M_i, with the smallest acyclic number in $TG \backslash TG(x)$ to a PN, where $\backslash$ denotes the difference of two sets. Only those PNs which have enough idle times to ensure the timely completion of M_i and survive the branching test will be considered as candidates for allocating M_i. The bounding test is then applied to those vertices expanded from x by allocating M_i to one of the candidate PNs. The UBOF, $\hat{P}_{ND}(y)$, of each newly-generated (intermediate) vertex y is calculated by scheduling modules $\in TG(y)$ with the **MSA** and evaluating $P_{ND1}(y)$ and $P_{ND2}(y)$ with the expressions derived in Sections 2.4 and 2.5, respectively. If a vertex y has its $\hat{P}_{ND}(y)$ greater than the currently best objective function value P^*_{ND}, it survives the bounding test, and will be made active and considered for vertex expansion in the next stage; otherwise, it will be pruned. The algorithm terminates when an optimal solution is

---

[6] Recall that we aim at *maximizing*, instead of minimizing, the objective function. Thus, an upper bound of the objective function should be derived for each intermediate vertex.

[7] $TG(x) = TG$ if x is a complete allocation.

found.

The **MAA** is outlined below. The **MSA** which schedules the modules assigned to each PN will be discussed in Section 2.4.1. The expressions of $P_{ND1}(y)$ and $P_{ND2}(y)$ will be derived in Sections 2.4.2 and 2.5, respectively. The branching and bounding tests used to achieve **BB** efficiency will be treated in Sections 2.6.1 and 2.6.2, respectively.

MA Algorithm:

Step 1. Generate the root, x_0, of the search tree, which corresponds to a null allocation. Set $AN := \{x_0\}$.

Step 2. Set $TG(x_0) := \emptyset$, $x_{opt} := x_0$, and the objective function value achieved by x_{opt}, $P^*_{ND} = 0.0 = \hat{P}_{ND}(x_0)$.

Step 3. While $AN \neq \emptyset$ **do**
/* an optimal allocation has not yet been found */

Step 3.1. Node Selection Rule:

Step 3.1.1. Select the vertex $x \in AN$ with the largest $\hat{P}_{ND}(x)$.

Step 3.1.2. If $\hat{P}_{ND}(x) < P^*_{ND}$, terminate the **MAA**, and x_{opt} is the optimal solution. Otherwise, set M_i to be the module $\in TG \backslash TG(x)$ with the smallest acyclic number, and $AN := AN \setminus \{x\}$.

Step 3.2. Branching Test:

Step 3.2.1. Conduct the branching test on each PN. Only those PNs which survive the branching test will be considered for allocating M_i (Section 2.6.1).

Step 3.2.2. Expand x by generating its valid child vertices, each of which corresponds to allocating M_i to one of the surviving PNs.

Step 3.3. Bounding Test: For each newly-generated vertex y,

Step 3.3.1. Use the **MSA** (Section 2.4.1) to find an optimal schedule for $TG(y)$ under y and calculate the UBOF, $\hat{P}_{ND}(y)$.

Step 3.3.2. If $\hat{P}_{ND}(y) \leq P^*_{ND}$, then prune y. Otherwise, the following two cases are considered:

Case 1. If y is a partial allocation, then set $AN := AN \cup \{y\}$, i.e., make y an active vertex.

Case 2. If y represents a complete assignment, $\hat{P}_{ND}(y)$ is the actual P_{ND} achieved under y. Since $\hat{P}_{ND}(y) > P^*_{ND}$, set $x_{opt} := y$ and $P^*_{ND} = \hat{P}_{ND}(y)$ to indicate that y has now become the best allocation found thus far.

2.4 Evaluation of Timeliness

In this section, we evaluate $P_{ND1}(x)$ for a given allocation x. We first consider the **MSA** which schedules all the modules assigned to a PN, say N_k, under x such that the maximum module tardiness is minimized subject to task release times and precedence constraints. By applying the **MSA** to each PN, we can obtain an optimal module schedule under x. Second, we calculate the probability, $P_{tc}(T_l \mid x)$, that a task T_l will be completed

before its deadline under allocation x and the corresponding optimal schedule obtained from the **MSA**. $P_{NDl}(x)$ can then be calculated from $P_{tc}(T_\ell \mid x)$, $\forall T_\ell$.

2.4.1 The Module Scheduling Algorithm

To facilitate the description and analysis of the **MSA**, we need to introduce the following notation:

- TG_c: a component task graph of TG. If TG contains loops or OR-subgraphs, it will be replaced by a set of component task graphs without loops and OR-graphs before applying the **MSA** (see Section 2.4.2 for more on this). For the time-being, we only need to know that TG_c contains neither loops nor OR-subgraphs.
- $TG_c(x)$: the set of modules $\in TG_c$ allocated under x.
- $S_k(x) = \{M_i : x_{ik} = 1\}$: the set of modules assigned to N_k under x.
- r_i: the release time of M_i, or the earliest time M_i can start its execution.
- LC_i: the latest completion time of M_i to ensure that all of its succeeding tasks will meet their deadlines.
- C_i: the completion time of M_i, which is determined by the **MSA**.
- e_i: the execution time of M_i.
- $\hat{e}_i$: the *modified* execution time of M_i, where

$$\hat{e}_i = \begin{cases} e_i & \text{if } M_i \text{ is scheduled to be executed upon its release at time } r_i \\ C_i - r_i & \text{otherwise.} \end{cases}$$

  $\hat{e}_i$ is used to include the effect of queueing M_i on the release times of all those modules that succeed M_i.
- $f_i(C_i)$: the cost incurred by completing M_i at C_i.
- $com_{ij}(x)$: the IMC time between M_i and M_j under x.
- d_{ij}: the IMC volume (measured in data units) between M_i and M_j.
- t_{mn}: the nominal delay (measured in time units per data unit) of link ℓ_{mn}.
- $n(k,\ell)$: the number of edge–disjoint paths between N_k and N_ℓ.
- $I(m,n,k,\ell)$: the indicator variable such that $I(m,n,k,\ell) = 1$ if ℓ_{mn} lies on one of the $n(k,\ell)$ edge–disjoint paths between N_k and N_ℓ.
- $Y_{k\ell} = \frac{1}{n(k,\ell)} \sum_{m=1}^{K_{pn}} \sum_{n=1}^{K_{pn}} I(m,n,k,\ell) \cdot t_{mn}$: the nominal delay (measured in time units per data unit) between N_k and N_ℓ.
- B: the minimal set of modules that are processed without any idle time in $[r(B), c(B))$, where $r(B) = \min_{M_i \in B} r_i$, $c(B) = r(B) + e(B)$, and $e(B) = \sum_{M_i \in B} e_i$.
- dg_i: the outdegree of M_i within a block of modules under consideration.

Specifically, $|S_k(x)|$ modules (possibly belonging to different tasks) are to be scheduled preemptively on N_k. Each module M_i becomes available upon its release at time r_i which is initially set to the invocation time of the task to which M_i belongs. Precedence

relations ($\rightarrow$) are considered in the entire task system: if $M_j \rightarrow M_i$ then M_i cannot start its execution before the completion of M_j, regardless whether M_i and M_j are assigned to the same PN or not. Execution of a module may be preempted and then resumed later. Associated with each M_i is a monotone nondecreasing cost function $f_i(C_i)$. We want to find a schedule for the modules in $S_k(x)$ such that $f_{max}(S_k(x)) \triangleq \max_{M_i \in S_k(x)} f_i(C_i)$ is minimized. The schedule with the minimal cost $f^*_{max}(S_k(x))$ is said to be an *optimal* schedule of $S_k(x)$.

Before proceeding to describe and analyze the **MSA**, we define the cost function $f_i(C_i)$ and discuss how to calculate the two parameters, LC_i and r_i, $\forall i$. The cost function is defined as

$$f_i(C_i) = C_i - LC_i, \tag{2.1}$$

where LC_i is the latest time M_i must be completed to ensure the timeliness of all of its succeeding modules, and C_i is the completion time of M_i determined by the **MSA**. If $C_i > LC_i$, a positive cost will occur. Thus, minimizing the maximum cost function is equivalent to minimizing the maximum tardiness of modules in TG_c.

The latest completion time, LC_i, of $M_i \in TG_c$ is obtained as follows. Let LC_i be initially set to the deadline of the task to which M_i belongs. Then, modify LC_i as

$$LC_i = \min\{LC_i, \min_j\{LC_j - e_j - com_{ij}(x) : M_i \rightarrow M_j\}\}, \quad i = N_M - 1, ..., 1, \tag{2.2}$$

where the modules are assumed to be numbered in acyclic order and

$$com_{ij}(x) = \begin{cases} 0, & \text{if } M_i \text{ and } M_j \text{ are assigned to the same PN under } x, \\ d_{ij}Y_{k\ell}, & \text{if } M_i \text{ and } M_j \text{ are assigned to } N_k \text{ and } N_\ell, \text{ respectively, under } x. \end{cases}$$

Note that Eq. (2.2) computes backward from $i = N_M - 1$ to $i = 1$, because M_N has no successor by the nature of acyclic order, and thus, the latest completion time of M_N is exactly the deadline of the task it belongs to. When x is a partial allocation and either M_i or M_j or both have not yet been assigned, $com_{ij}(x)$ may be left undefined. The rules used to (optimistically) estimate these com_{ij}'s will be given in Section 2.6.

The release time, r_i, of $M_i \in TG_c(x)$ is obtained as follows. Let r_i be initially set to the invocation time of the task to which M_i belongs. Then, modify r_i as

$$r_i = \max\{r_i, \max_j\{r_j + \hat{e}_j + com_{ji}(x) : M_j \rightarrow M_i\}\}, \quad 2 \leq i \leq N_M, \tag{2.3}$$

where r_1 is the invocation time of the task to which M_1 belongs, and $\hat{e}_j = \max\{C_j - r_j, e_j\}$ is the *modified* execution time which equals the sum of M_j's execution time, e_j, and M_j's queueing time (if M_j is not scheduled to be executed upon its release). $\hat{e}_j$ is used to include the effect of queueing M_i's preceding module, M_j, on M_i's release time.

Note that the modified execution times of all M_i's preceding modules must be available prior to the calculation of r_i. This is achieved by allocating the modules in the order of their acyclic numbers. When an intermediate vertex y survives the bounding test and is put in AN, all modules in $TG_c(y)$ will have been scheduled and their completion times (and thus modified execution times) will be determined in the bounding process (Step 3.3 in the **MAA** in Section 2.3). Thus, when x is expanded from its parent vertex y in the next stage by adding the new assignment of M_i, the schedules, completion times and modified execution times of all modules in $TG_c(y)$ (which includes all preceding modules of M_i) must have been determined. So, all the $\hat{e}_j$'s needed in Eq. (2.3) are known at the time of calculating r_i.

Fig. 2.5 shows an example of how r_i's and LC_i's are calculated.[8] The allocation x in Fig. 2.5 assigns M_1, M_3, M_5, and M_6 to N_1, and the other modules to N_2. Both module execution times and task deadlines are specified in the figure. The IPC delay is assumed to be 0.5 unit of time, i.e., $d_{ij}Y_{kl} = 0.5$ wherever applicable. For example, the release time r_4 of M_4 is calculated as

$$\begin{aligned} r_4 &= \max\{r_4, r_1 + \hat{e}_1 + com_{1\,4}(x), r_2 + \hat{e}_2 + com_{2\,4}(x), r_3 + \hat{e}_3 + com_{3\,4}(x)\} \\ &= \max\{0, 0+1+0.5, 0+1, 1.5+2+0.5\} = 4, \end{aligned}$$

and the latest completion time, LC_{12}, of M_{12} is calculated as

$$LC_{12} = \min\{LC_{12}, LC_{13} - e_{13} - com_{12\,13}(x)\} = \min\{10, 12-1\} = 10.$$

Now, we describe the **MSA**, the theoretical base of which is grounded on the result of [BLLK83]. First, we arrange the modules $\in S_k(x)$ in the order of nondecreasing release times. We then decompose $S_k(x)$ into blocks, where a block $B \subset S_k(x)$ is defined as the minimal set of modules processed without any idle time from $r(B) = \min_{M_i \in B} r_i$ until $c(B) = r(B) + e(B)$, where $e(B) = \sum_{M_i \in B} e_i$. That is, each $M_i \notin B$ is either completed no later than $r(B)$ or not released before $c(B)$. For example, as shown in Fig. 2.6, the set of modules assigned to N_1 in Fig. 2.5, $S_1(x) = \{M_1, M_3, M_5, M_6\}$, can be decomposed into three blocks, while the set of modules assigned to N_2, $S_2(x) = \{M_2, M_4, M_8, M_9, M_{10}, M_{11}, M_{12}, M_{13}\}$, can be decomposed into two blocks.

Obviously, scheduling modules in a block B is irrelevant to that in other blocks, so we can consider each block separately. Let dg_i denote the *outdegree* of M_i within B, i.e., the number of modules $M_j \in B$ such that $M_i \rightarrow M_j$. For each block B, we first determine the set $\hat{B} \triangleq \{M_i : M_i \in B, dg_i = 0\}$, i.e., modules without successors in B, and then select

[8] As will be discussed later in Section 2.4.2, the task graph in Fig. 2.5 is a component graph of the TG in Fig. 2.4.

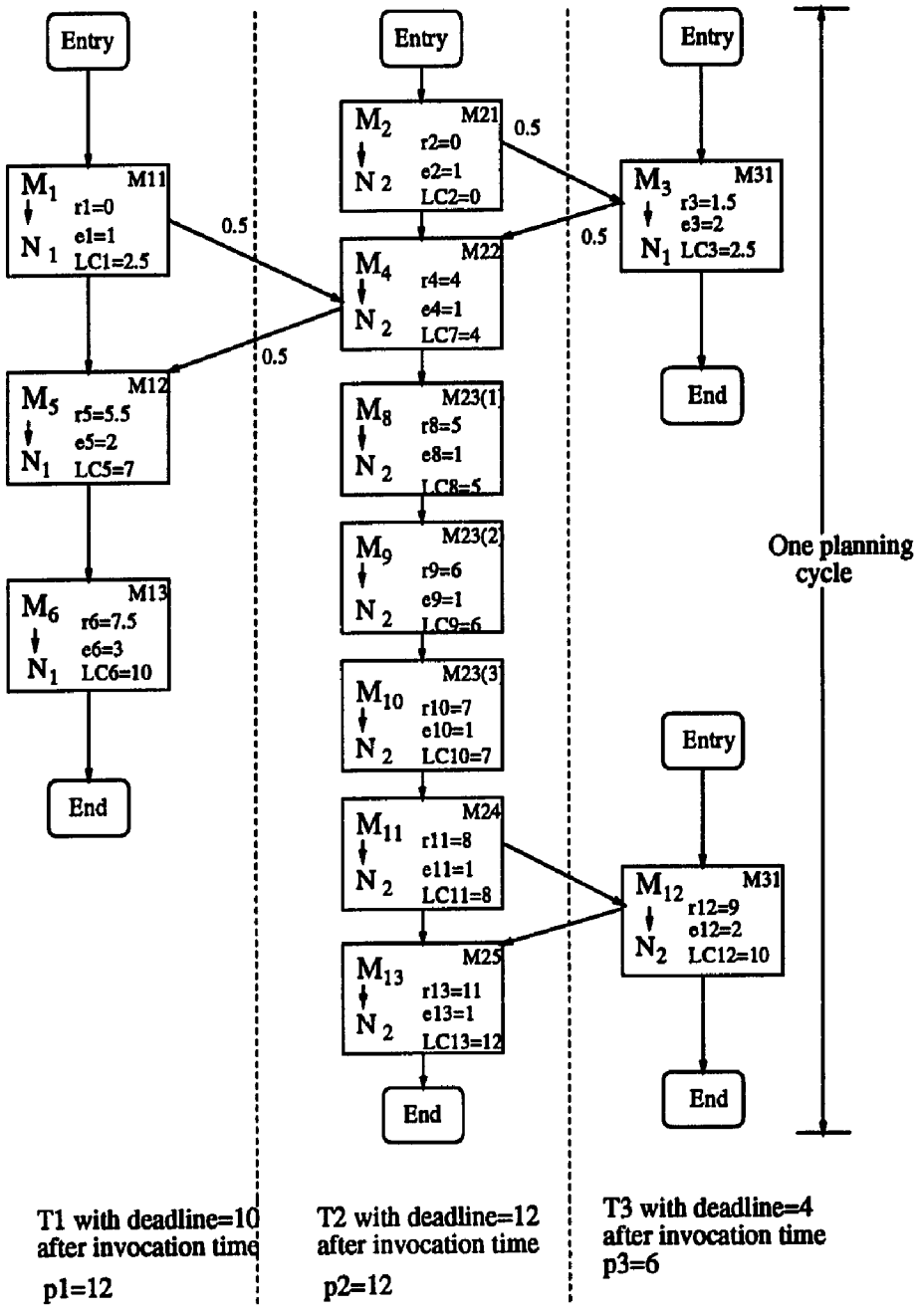


Figure 2.5: An example showing how r_i's and LC_i's are computed. All tasks are first invoked at time 0. (In this particular example, $\hat{e}_j = e_j$, $1 \leq j \leq 12$.)

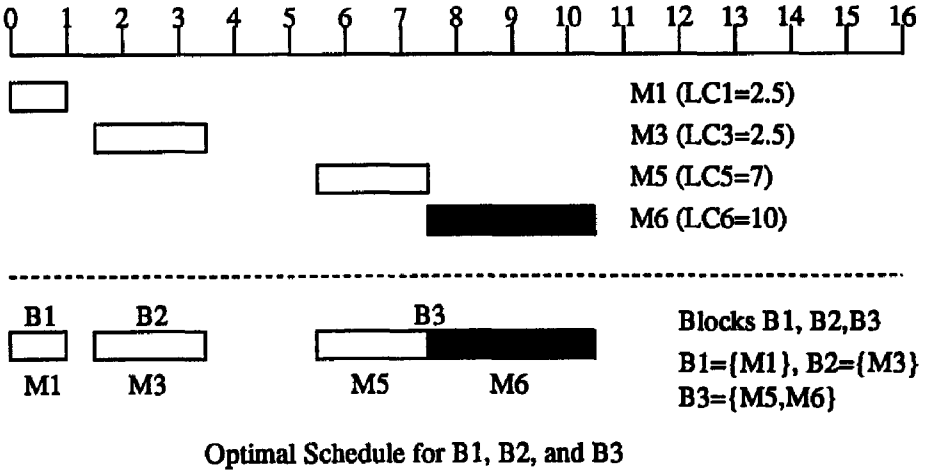


Figure 2.6: (a) Optimal schedule on N_1 under allocation x in which M_1, M_3, M_5, and M_6 are assigned to N_1, while the other modules are assigned to N_2.

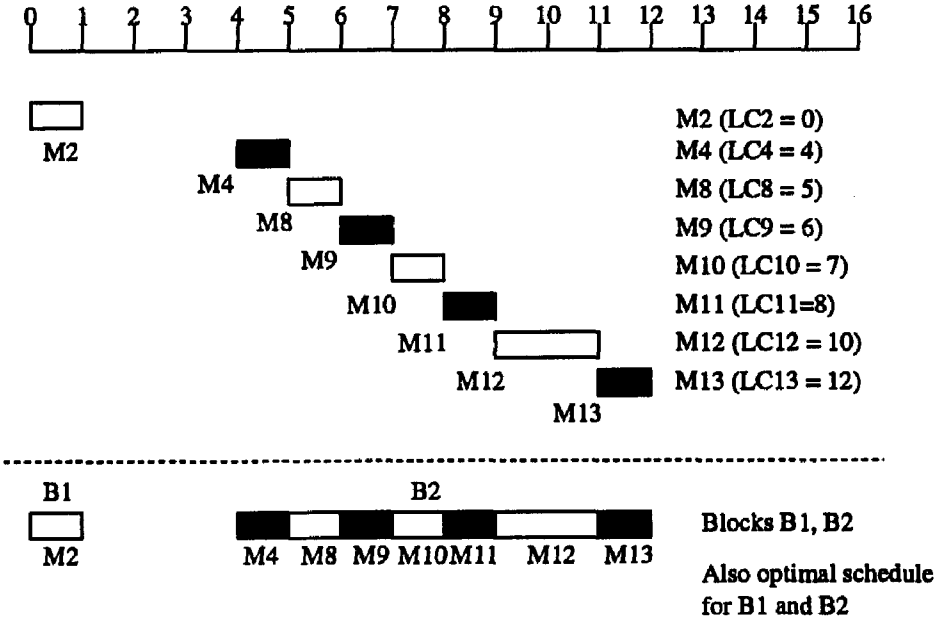


Figure 2.6: (b) Optimal schedule on N_2 under allocation x.

a module M_m such that

$$f_m(c(B)) = \min_{M_i \in \hat{B}} f_i(c(B)), \tag{2.4}$$

i.e., M_m has no successor within B and incurs a minimum cost if it is completed last in $\hat{B}$. (In case of a tie, we choose the module with the largest acyclic number.) Now, consider an optimal schedule for the modules in B subject to the restriction that M_m is processed only if no other module is waiting to be processed. This optimal schedule consists of two parts:

Sched1: An optimal schedule with the cost $f^*_{max}(B - \{M_m\})$ for the set $B - \{M_m\}$ which could be decomposed into a number of subblocks $\hat{B}_1, \hat{B}_2, ..., \hat{B}_{\hat{b}}$.

Sched2: A schedule for M_m, which is given by $[r(B), c(B)] - \cup_{j=1}^{\hat{b}} [r(\hat{B}_j), c(\hat{B}_j)]$, where $r(B) = \min_{M_i \in B} r_i$ and $c(B) = r(B) + e(B)$ with $e(B) = \sum_{M_i \in B} e_i$.

For this optimal schedule, we have

$$f^*_{max}(B) \text{ with the above restriction } = \max\{f_m(c(B)),\ f^*_{max}(B - \{M_m\})\} \leq f^*_{max}(B), \tag{2.5}$$

where the last inequality comes from:

1. $f^*_{max}(B) \triangleq \min \max_{M_i \in B} f_i(C_i) \geq \min_{M_i \in B} f_i(c(B)) = \min_{M_i \in \hat{B}} f_i(c(B)) = f_m(c(B))$ by the way $\hat{B}$ was constructed from B and Eq. (2.4).
2. Since $B - \{M_i\}$ is a subset of B, $f^*_{max}(B) \geq f^*_{max}(B - \{M_i\}), \forall M_i$.

It follows from Eq. (2.5) that there exists an optimal schedule in which M_m is scheduled only if no other module is waiting to be scheduled. By repeatedly and recursively applying the above procedure to each of the subblocks $\hat{B}_1$, $\hat{B}_2$,..., $\hat{B}_{\hat{b}}$, we obtain an optimal schedule for B. The rationale behind the **MSA** is that a PN is never left idle when there are modules ready to execute, and by virtue of the cost function defined, it is always the module M_i with the smallest LC_i that will be executed among all released modules.

Fig. 2.7 gives an illustrative example showing how the **MSA** schedules the modules assigned to a PN. r_i and LC_i, $1 \leq i \leq 5$, are assumed to have been computed from the entire task graph and are given in the figure. By ordering the modules according to their increasing release times, we obtain two blocks: $B_1 = \{M_1, M_2, M_3, M_4\}$ from $[0, 8]$ (i.e., $r(B_1) = 0$, $e(B_1) = 8$, and $c(B_1) = 8$) and $B_2 = \{M_5\}$ from $[9, 11]$ (i.e., $r(B_2) = 9$, $e(B_2) = 2$, and $c(B_2) = 11$). The schedule for B_2 is trivial, because B_2 consists of a single module and itself represents an optimal schedule for B_2. For B_1 we have $\hat{B}_1 = \{M_3, M_4\}$ and select M_3 to be processed only when no other modules are waiting since $LC_3 > LC_4$. Now $B - \{M_3\}$ consists of two subblocks: $B_{11} = \{M_1, M_2\}$ from $[0, 3]$ and $B_{12} = \{M_4\}$ from $[4, 6]$. B_{12}

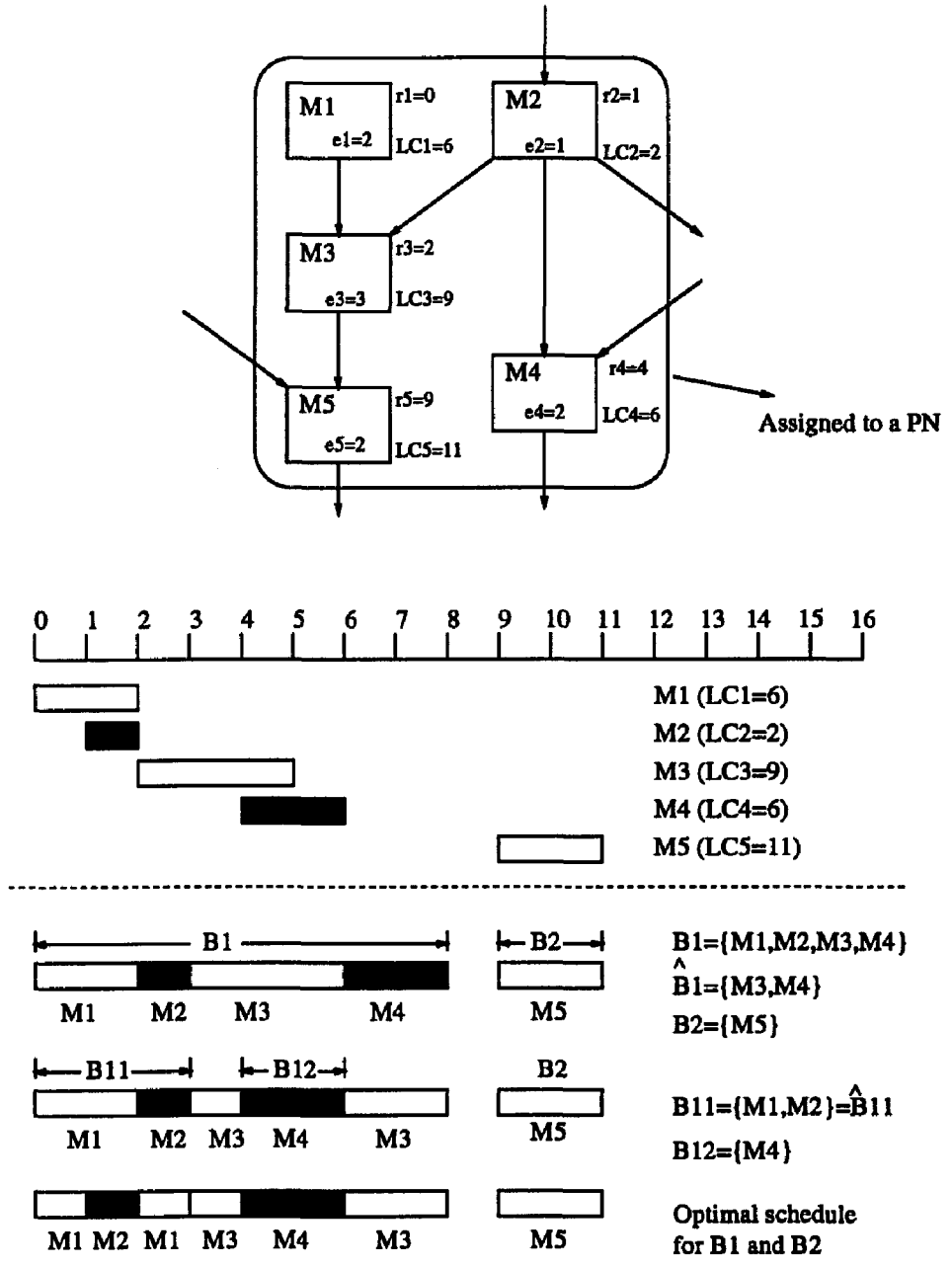


Figure 2.7: An example showing how the **MSA** schedules the modules assigned to a PN. Note that all modules make their latest completion times under the optimal schedule.

itself represents an optimal schedule. For B_{11} we have $\hat{B}_{11} = \{M_1, M_2\}$ and select M_1 to be processed last since $LC_1 > LC_2$. The final optimal schedule for B_1 is obtained by combining the optimal schedule for B_{11} and B_{12} (**Sched1**) and the schedule for M_3 (**Sched2**) which consists of $[0,8] - [0,3] \cup [4,6]$. The result is depicted in the last row of Fig. 2.7.

Take the task graph in Fig. 2.5 as another example, and consider the schedule on N_1. As shown in Fig. 2.6, $S_1(x)$ is composed of three blocks, $B_1 = \{M_1\}$, $B_2 = \{M_3\}$, and $B_3 = \{M_5, M_6\}$. The optimal schedule can be readily obtained (Fig. 2.6(a)), since the schedules for B_1 and B_2 are trivial, and for B_1 we have $\hat{B}_3 = \{M_6\}$, meaning that M_6 has to be processed only when no other module is waiting to be processed. The schedule on N_2 can be similarly obtained (Fig. 2.6(b)). The fact that not every module in Fig. 2.5 is completed before its latest completion time is due to the undesirable choice of module allocation. The schedules obtained above are, however, optimal under the *given* allocation and the given timing constraints. The **MSA** along with the time complexity in each step is summarized below.

MS Algorithm:

Step 1: Compute the latest completion time LC_i, $1 \leq i \leq N_M$, for TG_c. This computation requires $O(N_M^2)$ time.

Step 2: Compute the release time r_i for $M_i \in TG_c(x)$ with respect to their precedence constraints. This computation, in the worst case, requires $O(N_M^2)$ time.

Step 3: Construct the blocks B_1, B_2,..., B_b of $S_k(x)$ for every N_k by ordering the modules $\in S_k(x)$ according to their nondecreasing release times. This ordering requires $O(|S_k(x)| \cdot log|S_k(x)|)$ time, $\forall k$.

Step 4: For each block B_i, $1 \leq i \leq b$, update the outdegree, dg_j, of every $M_j \in B_i$. This update requires $O(|S_k(x)|^2)$ time for all B_i's $\subset S_k(x)$.

Step 5: For each block B_i, select $M_m \in B_i$ subject to Eq. (2.4), determine the subblocks of $B_i - \{M_m\}$,a nd construct the schedule for M_m as given in **Sched2**. Then, update the dg_j of every $M_j \in B_i - \{M_m\}$ with respect to the subblock of $B_i - \{M_m\}$ to which M_j belongs. By repeatedly applying **Step 5** to each of the subblocks of $B_i - \{M_m\}$, one can obtain an optimal schedule. The time complexity for all repeated applications of **Step 5** is bounded by $O(|S_k(x)|^3)$.

Since the time complexity associated with each step is polynomial, the **MSA** is a polynomial algorithm.

2.4.2 Calculation of $P_{ND1}(x)$

We discuss the calculation of $P_{tc}(T_t \mid x)$ under x and the schedules obtained from the **MSA** for every task T_t. The probability, $P_{ND1}(x)$, that no tasks miss their deadline under x can then be calculated from $P_{tc}(T_t \mid x)$, $\forall T_t$.

Conceptually, given TG and x, we can determine the set, $S_k(x)$, of modules $\in TG$ assigned to N_k and use the **MSA** to schedule modules in $S_k(x)$ $\forall k$. The completion time(s) of the last module(s) in $T_\ell \cap TG$ under these schedules determines whether T_ℓ can be completed in time or not. But it is not simple to implement the above concept, since TG may contain loops and/or OR-subgraphs. This leads to the difficulty of determining the release times, the latest completion times, and the execution times of modules, all of which are needed by the **MSA**. Moreover, one cannot determine which module of T_ℓ to execute last if the last component in T_ℓ is an OR-subgraph.

Component Graphs

To resolve the above difficulty, we first determine the latest completion time, LC_i, of $M_i \in TG$ using Eq. (2.2), assuming that

A1. Every OR-subgraph following M_i, if any, is viewed as an AND-subgraph by ignoring branching probabilities.

A2. Every loop L_a following M_i, if any, is replaced by a cascade of n_{L_a} copies of its loop body, where n_{L_a} is the maximum loop count.

A1 and **A2** ensure that there is enough CPU time left for the timely completion of OR-subgraphs and loops after M_i.

Second, we represent the TG with a set of component task graphs. Specifically, for each loop $L_a \in TG$ we define $TG_{L_a,m}$, $1 \leq m \leq n_{L_a}$, as the TG with L_a replaced by the cascaded m copies of its loop body. In $TG_{L_a,m}$, the last copy of $M_i \in L_a$ bears the LC_i calculated above, while the $(n_{L_a} - j)$-th copy of M_i bears the latest completion time $LC_i - j \cdot e(L_a)$, where $e(L_a)$ is the execution time of the loop body. With probability $(1-q_a)q_a^{m-1}$, the TG will be represented by $TG_{L_a,m}$, where q_a is the looping-back probability of L_a. Also, for each OR-subgraph $O_b \in TG$ we define $TG_{O_b,n}$, $1 \leq n \leq n_{O_b}$, as the TG with O_b replaced by its n-th branch, where n_{O_b} is the number of branches in O_b. With probability $q_{b,n}$, the TG will be represented by $TG_{O_b,n}$, where $q_{b,n}$ is the branching probability of the n-th branch of O_b.

Then, we represent the TG with the set of all possible combinations. For example, if there exists a loop L_a and an OR-subgraph O_b in TG, then there are a total of $n_{L_a} \times n_{O_b}$ component graphs of TG, and with probability $p_c = (1-q_a)q_a^{m-1} \cdot q_{b,n}$, the TG is represented by $TG_c = TG_{L_a,m;O_b,n}$, where $TG_{L_a,m;O_b,n}$ is the TG with L_a replaced by the cascaded m copies of its loop body and O_b replaced by its n-th branch. (One can trivially extend this to the case where there are more than one loop and/or OR-subgraph.) Hence, for example,

the TG in Fig. 2.4 can be represented by the set of 6 component task graphs shown in Fig. 2.8.[9] Also given in Fig. 2.8 are the probabilities that TG is represented by one of its component graphs.

For each component graph, TG_c, of TG, we then calculate the release time, r_i, of $M_i \in TG_c$ using Eq. (2.3). For example, the release time r_i (the latest completion time LC_i) under x is given in the upper (lower) right corner of the box representing M_i in Fig. 2.8. Using the r_i's and LC_i's determined above, we can apply Steps 3–5 in the **MSA** to find the best schedules for all modules in TG_c. Note that in a component graph TG_c, the release time, r_i, and the number of times M_i is executed are both fixed, making it possible to decompose $S_k(x)$ into blocks.

Calculation

We now calculate the probability, $P_{tc}(T_\ell \mid x)$, that a task $T_\ell \in TG$ is completed before its deadline under x. The parameters needed for this calculation are:

- p_c: the probability that TG is represented by TG_c.
- $\{TG_c\}$: the set of component graphs of TG.
- $\hat{T}_\ell \triangleq \{M_i : M_i \in T_\ell \cap TG_c,\ dg_i = 0$ with respect to $T_\ell \cap TG_c\}$: the set of modules without any successor in $T_\ell \cap TG_c$.
- D_i: the critical time of $M_i \in \hat{T}_\ell$. D_i is obtained in the same way as LC_i except that the precedence relations, $M_i \rightarrow M_j$ when $M_j \notin T_\ell$, are ignored. That is, let D_i be initially set to the deadline of T_ℓ to which M_i belongs. Then, it is modified as:

$$D_i = \min\{D_i,\ \min_{M_j \in T_\ell}\{D_j - e_j - com_{ij}(x) : M_i \rightarrow M_j\}\}, \quad i = N-1, N-2, ..., 1.$$

  D_i can be interpreted as the latest completion time of M_i for the timely completion of T_ℓ, to which M_i belongs. Obviously, $D_i \geq LC_i$.

The probability, $P_{tc}(T_\ell \mid TG_c, x)$, that T_ℓ is completed in time under x for a given TG_c is then calculated as:

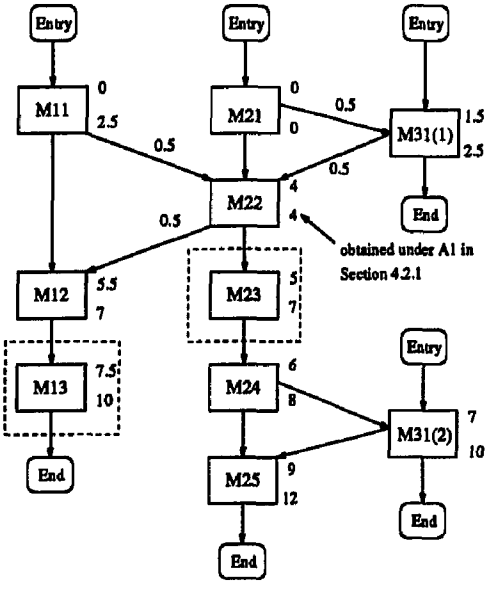
$$P_{tc}(T_\ell \mid TG_c, x) = \prod_{M_i \in \hat{T}_\ell} \delta(D_i - C_i), \tag{2.6}$$

where $\delta(\cdot)$ is the step function, i.e., $\delta(t) = 1$ for $t \geq 0$, and $\delta(t) = 0$ otherwise. Consequently, we have

$$P_{tc}(T_\ell \mid x) = \sum_{\{TG_c\}} p_c \cdot P_{tc}(T_\ell \mid TG_c, x), \tag{2.7}$$

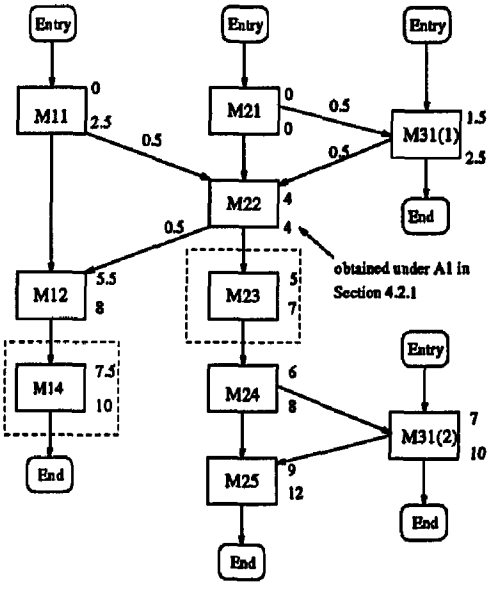
and,

$$P_{ND1}(x) = \prod_{\ell=1}^{N_T} P_{tc}(T_\ell \mid x). \tag{2.8}$$



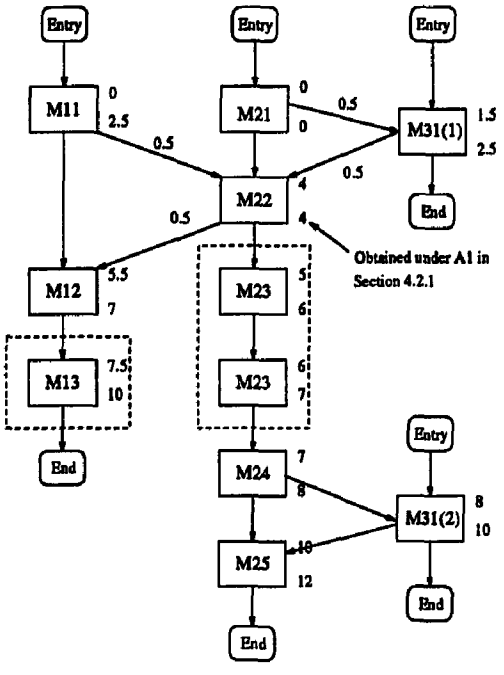
Representation of TG with the loop being replaced by its loop body and the OR-subgraph being replaced by the first branch.

$p_r = (1-p) \cdot q$



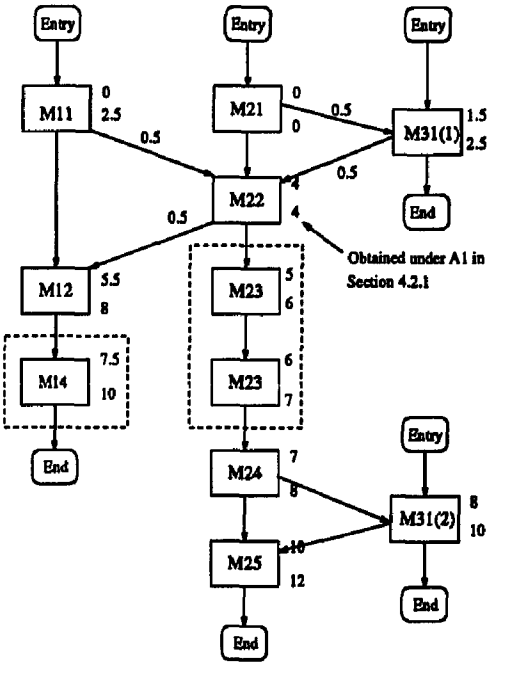
Representation of TG with the loop being replaced by its loop body and the OR-subgraph being replaced by the second branch.

$p_r = (1-p) \cdot (1-q)$



Representation of TG with the loop being replaced by the concatenation of 2 copies of the loop body and the OR-subgraph being replaced by the first branch.

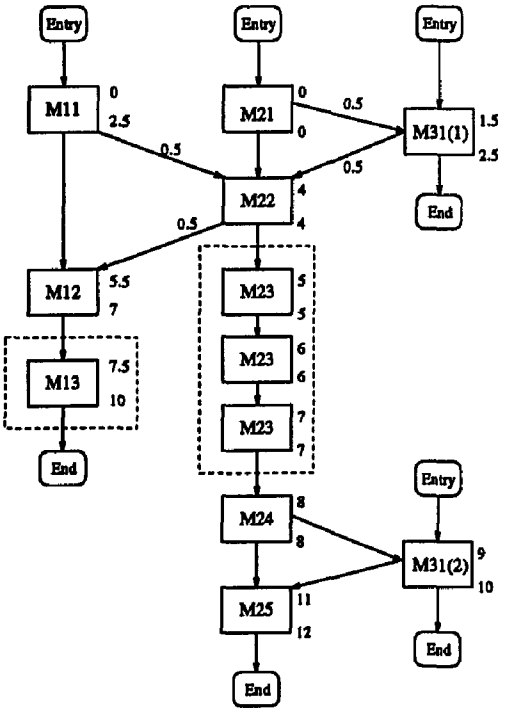
$p_r = (1-p) \cdot p \cdot q$



Representation of TG with the loop being replaced by the concatenation of 2 copies of the loop body and the OR-subgraph being replaced by the first branch.

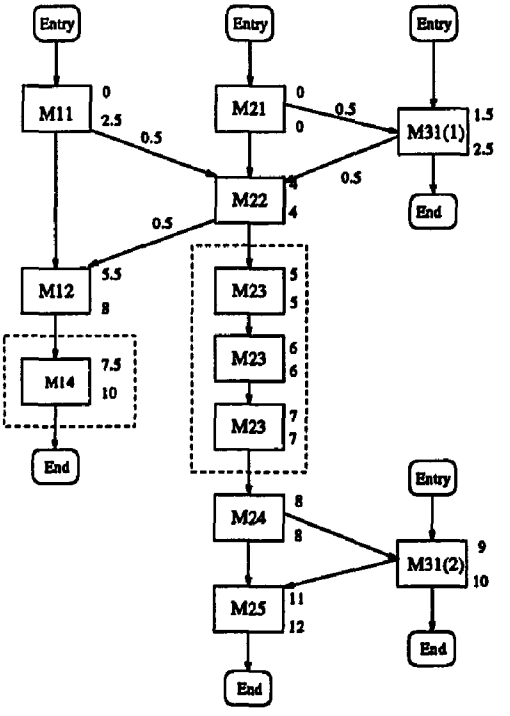
$p_r = (1-p) \cdot p \cdot (1-q)$

Figure 2.8: Component graphs of the TG in Fig. 2.4. Both release times and latest completion times are calculated under allocation x in which M_{11}, M_{12}, M_{13}, M_{14}, and $M_{31}(1)$ are assigned to N_1, while other modules are assigned to N_2.



Representation of TG with the loop being replaced by the concatenation of 3 copies of the loop body and the OR-subgraph being replaced by the first branch.

$p_r = p^2 \cdot q$



Representation of TG with the loop being replaced by the concatenation of 3 copies of the loop body and the OR-subgraph being replaced by the second branch.

$p_r = p^2 \cdot (1-q)$

Figure 2.8: (continued) Component graphs of the TG in Fig. 2.4.

Continuing with the example in Fig. 2.8, we have $\hat{T}_1 = \{M_{13}, M_{14}\}$, $\hat{T}_2 = \{M_{25}\}$, and $\hat{T}_3 = \{M_{31}(1), M_{31}(2)\}$ for all six component graphs, where $M_{31}(k)$ is the k-th invocation of M_{31}. The completion time C_i in $\hat{T}_1 \cup \hat{T}_2 \cup \hat{T}_3$ under x and the schedules obtained from the **MSA** are given in Table 2.1(a) along with the critical time, D_i, and latest completion time, LC_i. $P_{tc}(T_\ell \mid TG_c, x)$, $1 \le \ell \le 3$, $1 \le c \le 6$, can be readily calculated using Eq. (2.6), and are listed in Table 2.1(b). Using Eq. (2.7), we get

$$P_{tc}(T_1 \mid x) = (1-p)(1-q) + (1-p)p(1-q) + p^2(1-q) = (1-q),$$

$P_{tc}(T_2 \mid x) = 1$, and $P_{tc}(T_3 \mid x) = 1 - p^2$, where p and q, as denoted in Figs. 2.4 and 2.8, are the looping-back probability and the branching probability of the OR-subgraph, respectively. So, $P_{ND1}(x) = (1-q)(1-p^2)$.

[9] As one can readily check, the TG in Fig. 2.5 is exactly the 5-th component graph.

Task	Module M_i	LC_i	D_i	TG_1	TG_2	TG_3	TG_4	TG_5	TG_6
T_1	M_{13}	10	10	10.5	x	10.5	x	10.5	x
	M_{14}	10	10	x	9.5	x	9.5	x	9.5
T_2	M_{25}	12	12	10	10	11	11	12	12
T_3	$M_{31}(1)$	2.5	4	3.5	3.5	3.5	3.5	3.5	3.5
	$M_{31}(2)$	10	10	9	9	10	10	11	11

(a) Module completion times under allocation x.

$P_{tc}(T_t \mid TG_c, x)$	TG_1	TG_2	TG_3	TG_4	TG_5	TG_6
T_1	0	1	0	1	0	1
T_2	1	1	1	1	1	1
T_3	1	1	1	1	0	0

(b) $P_{tc}(T_t \mid TG_c, x)$

Table 2.1: Parameters needed to calculate $P_{ND1}(x)$ for the TG in Fig. 4.

2.5 Evaluation of Logical Correctness

In this section, we calculate the probability, $P_{ND2}(x)$, that: (i) all PNs are operational during the execution of modules assigned to them, and (ii) all communication links are operational during the course of IPCs. The derivation of $P_{ND2}(x)$ is similar to that of the reliability function in [SW89], but we relax the following two unrealistic assumptions used in [SW89]: (A1) the network topology is cycle-free, i.e., there is one and only one path between any pair of PNs; (A2) each module is executed only once in a task invocation.

Instead of the first assumption, we allow an arbitrary network topology, and also allow the IPCs between N_k and N_ℓ to take place over one of the (arbitrarily chosen) edge–disjoint paths between the two PNs. In contrast to the second assumption, we allow modules to be contained in loops and/or branches of OR-subgraphs, i.e., modules may be executed more than once, or not executed at all in a task invocation.

To facilitate the derivation of $P_{ND2}(x)$, we need the following notation:

- LP: the set of modules which are contained in loops.
- OR: the set of modules which are on the branches of OR-subgraphs.
- q_a: the looping-back probability of loop L_a.
- $q_{b,\ell}$: the branching probability of the ℓ-th branch of an OR-subgraph, O_b.
- n_{L_a}: the maximum count of loop L_a.
- n_{O_b}: the number of branches in an OR-subgraph O_b.
- λ_k: the constant exponential failure rate of N_k.

- $\hat{\lambda}_{mn}$: the constant exponential failure rate of link ℓ_{mn}. Failure occurrences are assumed to be statistically independent of one another.
- $P_{IMC}(i,j,n_c)$: the probability that the IMC between M_i and M_j occurs n_c times in one task invocation.
- $R_{mn}(i,j,n_c,x)$: the probability that link ℓ_{mn} is operational during the n_c occurrences of IMC between M_i and M_j under allocation x.
- $R_{\rm pn}(x)$: the probability that all PNs are operational during the execution of modules assigned to them under x.
- $R_{\rm link}(x)$: the probability that all links are operational for all IMCs under x.

Under allocation x, the probability that all PNs remain fault-free during the execution of the modules assigned to them is:

$$\begin{aligned} R_{\rm pn}(x) \;=\; & \{ \prod_{M_i \notin LP \cup OR} \prod_{k=1}^{K_{pn}} exp(-\lambda_k x_{ik} e_i) \} \cdot \\ & \{ \prod_{L_a \in LP} \sum_{\ell=1}^{n_{L_a}} [\; (1-q_a) q_a^{\ell-1} \cdot \prod_{M_i \in L_a} \prod_{k=1}^{K_{pn}} exp(-\lambda_k x_{ik} \ell e_i) \;] \} \cdot \\ & \{ \prod_{O_b \in OR} \sum_{\ell=1}^{n_{O_b}} [\; q_{b,\ell} \cdot \prod_{\substack{M_i \in \ell\text{-th} \\ \text{branch of } O_b}} \prod_{k=1}^{K_{pn}} exp(-\lambda_k x_{ik} e_i) \;] \}. \end{aligned} \tag{2.9}$$

Note that all factors except the one associated with $x_{ik} = 1$ in the term $\prod_{k=1}^{K_{pn}} exp(-\lambda_k x_{ik} e_i)$ reduce to 1. The expression within the first pair of braces is the probability that the PNs on which stand-alone modules[10] reside are operational during the execution of these modules. Similarly, the expression in the second (third) pairs of braces is the probability that the PNs on which the modules in loops (OR-subgraphs) reside are operational during the execution of these modules. In case M_i is contained in a loop L_a, with probability $q_a^{\ell-1}(1-q_a)$, M_i requires an execution time $\ell \cdot e_i$, and in case M_i is on the ℓ-th branch of an OR-subgraph, O_b, with probability $q_{b,\ell}$, M_i will be executed. Note that Eq. (2.9) can be readily extended to the case where a loop/OR–subgraph is contained in other loops and/or OR-subgraphs.

Consider Fig. 2.4 as an example, where $LP = \{M_8\}$ and $OR = \{M_6, M_7\}$. If all modules of T_1 are assigned to N_1 and all modules of T_2 and T_3 are assigned to N_2, i.e., $y_{11} = y_{51} = y_{61} = y_{71} = 1$ and $y_{22} = y_{32} = y_{42} = y_{82} = y_{92} = y_{10\,2} = y_{11\,2} = 1$, then

$$R_{\rm pn}(y) \;=\; e^{-\lambda_1(e_1+e_5)} \cdot e^{-\lambda_2(e_2+e_3+e_4+e_9+e_{10}+e_{11})} \cdot \{ q e^{-\lambda_1 e_6} + (1-q) e^{-\lambda_1 e_7} \} \cdot \sum_{\ell=1}^{n_L} (1-p) p^{\ell-1} e^{-\lambda_2 \ell e_8}.$$

The expression of $R_{\rm link}(x)$ calls for the derivation of $R_{mn}(i,j,n_c,x)$ and $P_{IMC}(i,j,n_c)$. $R_{mn}(i,j,n_c,x)$ is the probability that link ℓ_{mn} is operational during the n_c occurrences of

[10] modules which are contained in neither loops nor OR-subgraphs in the TG.

IMC between M_i and M_j under x, and can be expressed as:

$$R_{mn}(i,j,n_c,x) = \prod_{k=1}^{K_{pn}} \prod_{\ell=1,\ell\neq k}^{K_{pn}} exp(-\hat{\lambda}_{mn} \cdot \frac{n_c t_{mn} d_{ij}}{n(k,\ell)} \cdot x_{ik} x_{j\ell} \cdot I(m,n,k,\ell)). \tag{2.10}$$

Two remarks are in order:

- All the $K_{pn}(K_{pn}-1)$ terms in Eq. (2.10) — except for the term corresponding to $x_{ik} = x_{j\ell} = 1$ — reduce to 1.
- If M_i is assigned to N_k ($x_{ik} = 1$), M_j is assigned to N_ℓ ($x_{j\ell} = 1$), and ℓ_{mn} lies on one of the edge–disjoint paths between N_k and N_ℓ ($I(m,n,k,\ell) = 1$), then

$$R_{m,n}(i,j,n_c,x) = exp(-\hat{\lambda}_{mn} \frac{n_c t_{mn} d_{ij}}{n(k,\ell)}),$$

  where $\frac{n_c t_{mn} d_{ij}}{n(k,\ell)}$ is the average communication time over link ℓ_{mn} contributed by the n_c occurrences of IMC between M_i and M_j.

For example, given the simple distributed system represented by a complete graph of 3 PNs (where $n(k,\ell) = 2$ and $I(m,n,k,\ell) = 1$, $1 \leq m,n \leq 3$, $m \neq n$, $1 \leq k,\ell \leq 3$, $k \neq \ell$) and the TG in Fig. 2.4, we have

$$R_{12}(i,j,n_c,x) = exp(-\hat{\lambda}_{12} \cdot \frac{n_c t_{12} d_{ij}}{2} \cdot (x_{i1}x_{j2} + x_{i1}x_{j3} + x_{i2}x_{j1} + x_{i2}x_{j3} + x_{i3}x_{j1} + x_{i3}x_{j2})).$$

Under the allocation y in which modules of T_1 and $T_2 \cup T_3$ are assigned to N_1 and N_2, respectively, $R_{12}(1,4,n_c,y) = exp(-\hat{\lambda}_{12} \cdot \frac{n_c t_{12} d_{14}}{2})$ since $y_{11}y_{42} = 1$, and $R_{12}(2,4,n_c,y) = 1$, i.e., the IMCs between M_2 and M_4 are accomplished via shared memory and do not use link ℓ_{12} at all.

$P_{IMC}(i,j,n_c)$ is the probability that the IMC between M_i and M_j occurs n_c times in one task invocation. Obviously, $P_{IMC}(i,j,n_c) \neq 0$, $n_c \geq 1$, holds only when M_i and M_j communicate with each other, i.e., $d_{ij} > 0$. Specifically, in the case of $d_{ij} > 0$, we have the following conditions:

C1. $P_{IMC}(i,j,1) = 1$ and $P_{IMC}(i,j,n_c) = 0$ for $n_c > 1$ if neither of M_i and M_j resides in a loop or an OR-subgraph.

C2. $P_{IMC}(i,j,1) = 1$ and $P_{IMC}(i,j,n_c) = 0$ for $n_c > 1$ if one of M_i or M_j resides in a loop, while the other (not contained in an OR–subgraph) resides immediately before or after the loop.

C3. $P_{IMC}(i,j,n_c) = q_a^{n_c-1}(1-q_a)$, for $1 \leq n_c \leq n_{L_a} - 1$, and $P_{IMC}(i,j,n_{L_a}) = q_a^{n_{L_a}}$ if both M_i and M_j reside in the body of a loop L_a.

C4. $P_{IMC}(i,j,1) = q_{b,\ell}$ and $P_{IMC}(i,j,n_c) = 0$ for $n_c > 1$ if either M_i or M_j resides on the ℓ-th branch of an OR-subgraph, O_b, while the other resides immediately before or after O_b.

C5. $P_{IMC}(i,j,1) = q_{b,\ell}$ and $P_{IMC}(i,j,n_c) = 0$ for $n_c > 1$ if both M_i and M_j reside on the ℓ-th branch of an OR-subgraph, O_b.

Let $C_k \triangleq \{(M_i, M_j) : M_i$ and M_j satisfy condition Ck and $d_{ij} > 0\}$, $k = 1, 2$; $C_3(L_a) \triangleq \{(M_i, M_j) :$ both M_i and M_j reside in the loop body of L_a and $d_{ij} > 0\}$, $\forall\ L_a \in LP$; and $C_4(O_b, \ell) \triangleq \{(M_i, M_j) :$ either M_i or M_j or both reside on the ℓ-th branch of O_b and $d_{ij} > 0\}$, $\forall\ O_b \in OR$, $1 \le \ell \le n_{O_b}$. $R_{\text{link}}(x)$ can then be expressed as:

$$\begin{aligned} R_{\text{link}}(x) &= \{ \prod_{(M_i,M_j)\in C_1\cup C_2} \prod_{\ell_{mn}} R_{mn}(i,j,1,x) \} \cdot \\ &\quad \{ \prod_{L_a\in LP} \sum_{n_c=1}^{n_{L_a}} q_a^{n_c-1}(1-q_a) \cdot [\prod_{(M_i,M_j)\in C_3(L_a)} \prod_{\ell_{mn}} R_{mn}(i,j,n_c,x) \ ] \} \cdot \\ &\quad \{ \prod_{O_b\in OR} \sum_{\ell=1}^{n_{O_b}} q_{b,\ell} \cdot [\prod_{(M_i,M_j)\in C_4(O_b,\ell)} \prod_{\ell_{mn}} R_{mn}(i,j,1,x) \ ] \}. \end{aligned} \tag{2.11}$$

For clarity of presentation, Eq. (2.11) excludes the case where a loop/OR-subgraph is contained in other loops and/or OR-subgraphs. It is straightforward to extend Eq. (2.11) to include such a case.

Consider again the example of allocating the TG in Fig. 2.4 to the distributed system represented by a 3-complete graph: $C_1 = \{ (M_1, M_4), (M_1, M_5), (M_2, M_3), (M_2, M_4), (M_3, M_4), (M_4, M_5), (M_9, M_{10}), (M_9, M_{11}), (M_{10}, M_{11}) \}$, $C_2 = \{ (M_4, M_8), (M_8, M_9) \}$, $C_3 = \emptyset$, $C_4(O_1, 1) = \{ (M_1, M_6) \}$, and $C_4(O_1, 2) = \{ (M_1, M_7) \}$.

$$R_{\text{link}}(x) = \{ \prod_{(M_i,M_j)\in C_1\cup C_2} \prod_{\ell_{mn}} R_{mn}(i,j,1,x)\} \cdot \{ q \prod_{\ell_{mn}} R_{mn}(1,6,1,x) + (1-q) \prod_{\ell_{mn}} R_{mn}(1,7,1,x)\}.$$

Under the allocation y given in the previous example, we get

$$\begin{aligned} R_{\text{link}}(y) &= \prod_{\ell_{mn}} R_{mn}(1,4,1,y) \cdot \prod_{\ell_{mn}} R_{mn}(4,5,1,y) \\ &= e^{-(\hat{\lambda}_{12}t_{12}+\hat{\lambda}_{13}t_{13}+\hat{\lambda}_{32}t_{32})d_{14}/2} \cdot e^{-(\hat{\lambda}_{12}t_{12}+\hat{\lambda}_{13}t_{13}+\hat{\lambda}_{32}t_{32})d_{45}/2}, \end{aligned}$$

where the first (second) factor in the last expression is contributed by the IMC between M_1 and M_4 (between M_4 and M_5), e.g., $e^{-\hat{\lambda}_{12}t_{12}d_{14}/2}$ is contributed by the IPC between M_1 and M_4 which runs through ℓ_{12}, and $e^{-(\hat{\lambda}_{32}t_{32}+\hat{\lambda}_{13}t_{13})d_{14}/2}$ is contributed by the IPC which routes through ℓ_{13} and ℓ_{32}. Finally, we have

$$P_{ND2}(x) = R_{\text{pn}}(x) \cdot R_{\text{link}}(x). \tag{2.12}$$

2.6 Branching and Bounding Tests

The branching test uses the dominance relation derived from the requirement of timely completion of tasks to limit the number of child vertices generated in the branching process. The bounding test derives an upper bound of the objective function (UBOF) for each intermediate vertex with which one decides whether or not to prune an intermediate vertex in the bounding process.

2.6.1 Branching Test

Recall that expanding an intermediate vertex x in the search tree corresponds to allocating the module with the smallest acyclic number that has not yet been allocated (so, a module in $TG \setminus TG(x)$). The branching test uses the following dominance relation. M_i can be invoked after all its precedence constraints are met and must be completed by its latest completion time, LC_i, to ensure that all its succeeding tasks meet their deadlines. Hence, if (1) the idle time of a PN, say N_k, during the interval $[r_i, LC_i]$ is smaller than e_i, and (2) the module, say M_j, scheduled to be executed last[11] on N_k in $[r_i, LC_i]$ under a partial allocation x has tighter timing constraints than M_i (so no preemption on N_k to ensure the completion of M_i before LC_i), then allocating M_i to N_k is likely to miss M_i's latest completion time. Thus, N_k should not be a candidate PN for allocating M_i, i.e., fails the branching test.

Branching Test:

Step 1. Calculate optimistic estimates, r_i^o and LC_i^o, of r_i and LC_i, assuming that

- **A1.** Every pair of communicating modules that have not yet been assigned (i.e., $\in TG \setminus TG(x)$) reside on the same PN. That is, the IMC communication times in $TG \setminus TG(x)$ are set to zero.
- **A2.** The OR-subgraph preceding or following M_i, if any, is replaced by the branch with the *smallest* flowtime.
- **A3.** The loop preceding, containing, or following M_i, if any, is replaced by its loop body (i.e., the loop executes only once).

Step 2. Calculate an pessimistic estimate, LC_i^p, of LC_i, assuming that

- **A4.** The IMCs in $TG \setminus TG(x)$ are executed on N_k and N_ℓ with the largest nominal inter-PN delay $Y_{k\ell}$.
- **A5.** The OR-subgraph following M_i is replaced by the branch with the *largest* flowtime.
- **A6.** The loop following M_i (if any) is replaced by the cascaded n_L copies of its loop body, where n_L is its maximum loop count.

Step 3. For each N_k, check whether the following two conditions are true or not:

- **C1.** The idle time of N_k in $[r_i^o, LC_i^o]$ is less than e_i.
- **C2.** $LC_j \leq LC_i^p$, where LC_j is the latest completion time of the module, M_j, scheduled last in $[r_i^o, LC_i^o]$ on N_k under the partial allocation x.

If both conditions are true, then N_k fails the test and is not considered for allocating M_i.

---

[11] By 'last,' we mean the module is executed only if no other modules are waiting for processing on N_k.

A1–A3 ensure $r_i^o \leq r_i$ and $LC_i^o \geq LC_i$, thus making the interval $[r_i^o, LC_i^o]$ larger than $[r_i, LC_i]$. **A4–A6** ensure $LC_i^p \leq LC_i$, making M_i likely to preempt other modules on N_k. Consequently, the use of the optimistic interval, $[r_i^o, LC_i^o]$, and the pessimistic value, LC_i^p, ensures that the PNs which fail the branching test cannot indeed complete M_i in time.

2.6.2 Calculation of an UBOF for the Bounding Test

The bounding test calculates an UBOF for each intermediate vertex, and prunes (keeps) the intermediate vertex when the calculated UBOF $\leq$ ($>$) the best objective function, P_{ND}^*, found thus far. The bounding test uses the following principles. A vertex y is generated from its parent vertex x by adding the assignment $M_i \rightarrow N_k$ to the partial allocation x for some N_k that survives the branching test. After including $M_i \rightarrow N_k$, N_k needs to re-schedule the modules assigned to it under y (i.e., the modules $\in S_k(y)$) using the **MSA**. Because modules are assigned in acyclic order, all preceding modules of M_i in $S_k(y)$ have their latest completion times $< LC_i$, and their schedules will not be changed by the addition of M_i. On the other hand, if some non-preceding module, M_j, of M_i does change its schedule as a result of $M_i \rightarrow N_k$, then the release time(s) of all M_j's succeeding module(s) have to be changed accordingly. Consequently, the PNs ($\neq N_k$) on which these succeeding modules of M_j reside need to reconsider their module schedules.

Fig. 2.9 gives an example of how adding $M_i \rightarrow N_k$ to a partial allocation might affect the schedules on other PNs. In the partial allocation x prior to the assignment $M_6 \rightarrow N_1$, M_1 and M_2 are assigned to N_1, and M_3, M_4, and M_5 are assigned to N_2. The optimal schedules on N_1 and N_2 for x (obtained from the bounding process of the last stage) are shown in Fig. 2.9(a). Now, assign M_6 to N_1 to get the child vertex, y, of x. As shown in Fig. 2.9(b), N_1 needs to re-schedule its assigned modules. (Note that the schedule for M_1, however, does not change since M_1 is a preceding module of M_6.) Also, the schedule change on N_1 — especially, the schedule change for M_2 — alters the release times of M_2's succeeding modules, M_4 and M_5. So, the schedule on N_2, the PN on which M_4 and M_5 reside, needs to be changed as well (Fig. 2.9(b)).

Let $\widehat{PN}$ denote the set of PNs which need to reconsider their module schedules as a result of $M_i \rightarrow N_k$. Then an UBOF is calculated by the following steps.

Calculation of an UBOF:

Step 0. Represent the TG with the set, $\{TG_c\}$, of component graphs by using the method in Section 2.4.2.

Step 1. Calculate $\hat{P}_{ND1}(y)$ as follows:

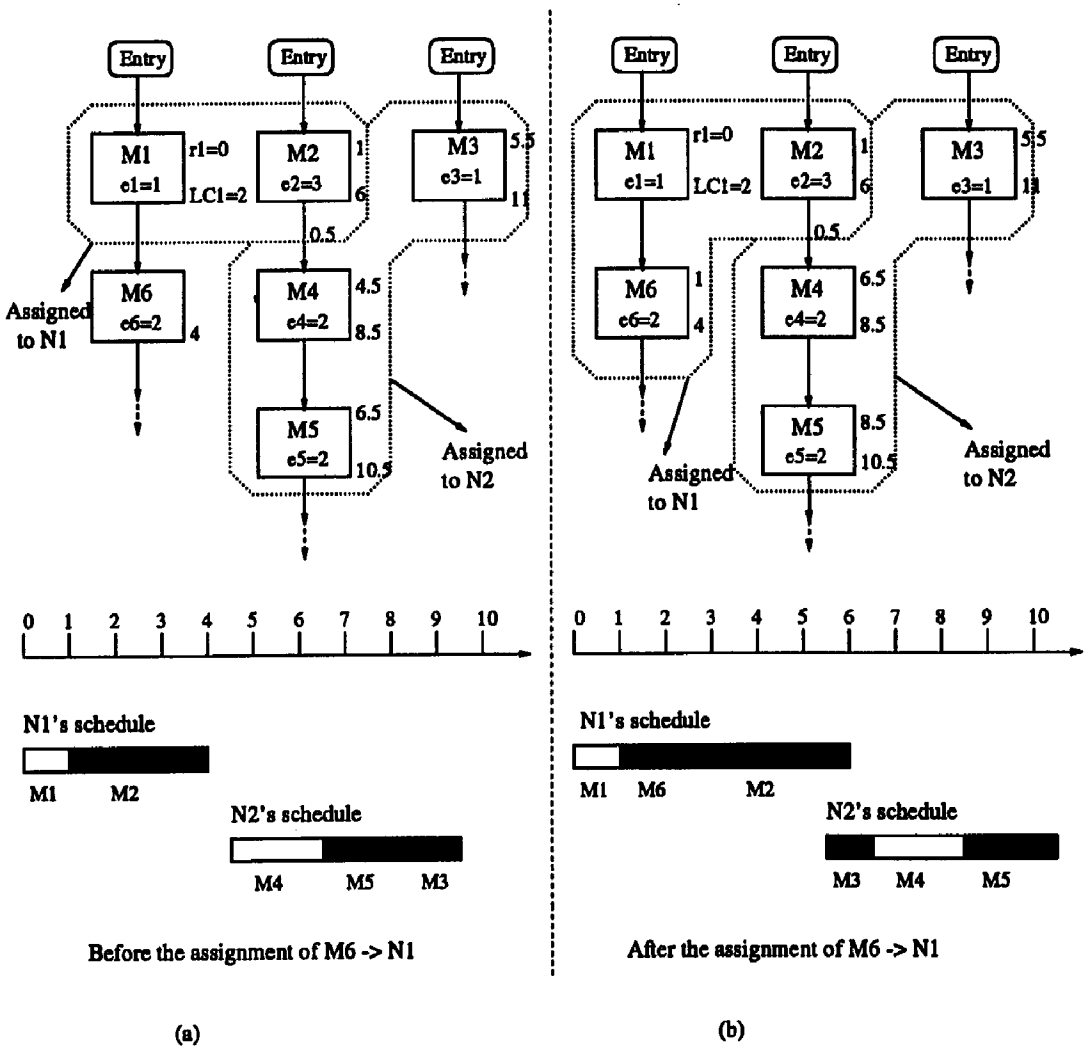


Figure 2.9: An example showing how a new assignment $M_i \rightarrow N_k$ might affect the module schedule of $N_m \neq N_k$.

Step 1.1. In each component graph TG_c:

Step 1.1.1 For every $N_m \in \widehat{PN}$, re-schedule the modules $\in S_m(y)$ by using the **MSA** and **A1** in the branching test.

Step 1.1.2. Calculate $P_{tc}(T_\ell \mid TG_c, y)$, $\forall T_\ell$, by using Eq. (2.6), where $\hat{T}_\ell$ in Eq. (2.6) is modified as

A7. $\hat{T}_\ell \triangleq \{M_i : M_i \in T_\ell \cap TG_c(y), dg_i = 0 \text{ w.r.t. } T_\ell \cap TG_c(y)\}$,
i.e., $TG_c(y)$ replaces TG_c in Eq. (2.6), and is the set of modules $\in TG_c$ allocated under y.

Step 1.2. Calculate $P_{tc}(T_\ell \mid y)$, $\forall T_\ell$, and $\hat{P}_{ND1}(y)$ by using Eqs. (2.7) and (2.8).

Step 2. Calculate $\hat{P}_{ND2}(y)$ by using Eqs. (2.9), (2.11) and (2.12), and

A8. Every $M_j \in TG \setminus TG(y)$ is assumed to be allocated to the most reliable PN, and every pair of communicating modules in $TG \setminus TG(y)$ reside on the same PN.

Step 3. Calculate $\hat{P}_{ND}(y) = \hat{P}_{ND1}(y) \cdot \hat{P}_{ND2}(y)$.

Note that $\hat{P}_{ND1}(y)$ derived above is an upper bound of P_{ND1} of any leaf vertex (complete allocation) generated from y because of **A1** and **A7**. When calculating $\hat{P}_{ND1}(y)$ we can use **A7** to exclude the case of whether or not modules $\in TG \setminus TG(y)$ can meet their latest completion times.

2.7 Numerical Examples

The performance of the **MAA** is evaluated according to the following sequence: (1) discussion on the generation of task graphs and distributed systems; (2) the characteristics of the **MAA**; (3) the practicality of the **MAA**.

2.7.1 Generation of Task Graphs and Distributed Systems

There are a large number of parameters that may affect the performance of the **MAA**. They can be classified as *system parameters* which specify the distributed system under consideration and *task parameters* which specify the TG. The generation of realistic TGs and distributed systems largely depends on how these parameters are specified. However, little is reported in the literature about "typical" real-time TGs and their communication patterns. Thus, we randomly generate both system and task parameters in our numerical experiments. The number of PNs in the distributed system is varied from 3 to 40, and the network topology is arbitrarily generated. The nominal delay, t_{mn}, associated with ℓ_{mn} is exponentially distributed with mean $0.1\overline{e}$, where $\overline{e}$ is the mean module execution time. The node failure rate, λ_i, and the link failure rate, $\hat{\lambda}_{mn}$, were varied from 10^{-5} to 0.5. The number of modules to be allocated is varied from 4 to 50. The execution time

of a module is exponentially distributed with mean 1.0 unit of time. The IMC volume between two communicating modules is uniformly distributed over $(0, 10]$ data units. The precedence constraints and the timing requirements of the TG are also randomly generated.

Before running experiments, we eliminated the TGs which were definitely infeasible. Infeasibility is detected by calculating release times and latest completion times of all modules, while ignoring all IMC times. If the interval between the latest completion time and the release time is less than the execution time for some module(s) in all the component graphs of a TG, this TG is infeasible (in the sense that some tasks cannot be completed in time even if infinite resources were available) and is not considered any further.

All experiments were performed on a SPARC station running the SUNOS 4.1.2 operating system. Due to space limitation, we present only a few representative solutions and statistical results. However, the conclusions drawn from the following summary were corroborated by all the experiments conducted.

2.7.2 Characteristics of the MAA

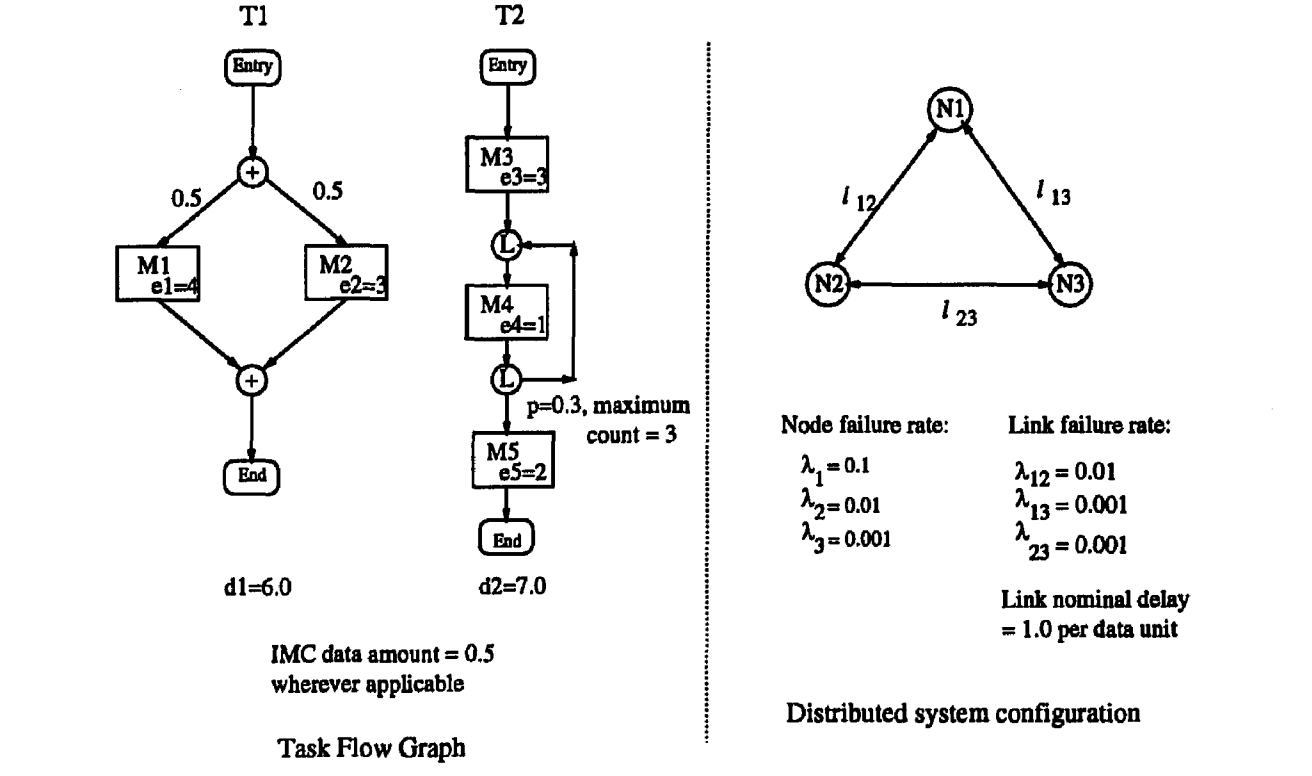
By virtue of the B&B method, the **MAA** always yields the best allocation. Fig. 2.10 shows an example of how vertices were visited by the **MAA** in the state-space search tree before the optimal vertex was found. The TG, the network topology, and their attributes used in the example are also given in Fig. 2.10. Only 34 vertices in the search tree (of 728 vertices) were visited before locating the best allocation.

To further examine the characteristics of the optimal allocation found by the **MAA**, experiments were performed on (1) TGs with different degrees of parallelism;[12] (2) task sets with different degrees of deadline tightness; (3) distributed systems with different nominal link delays and node/link failure rates.

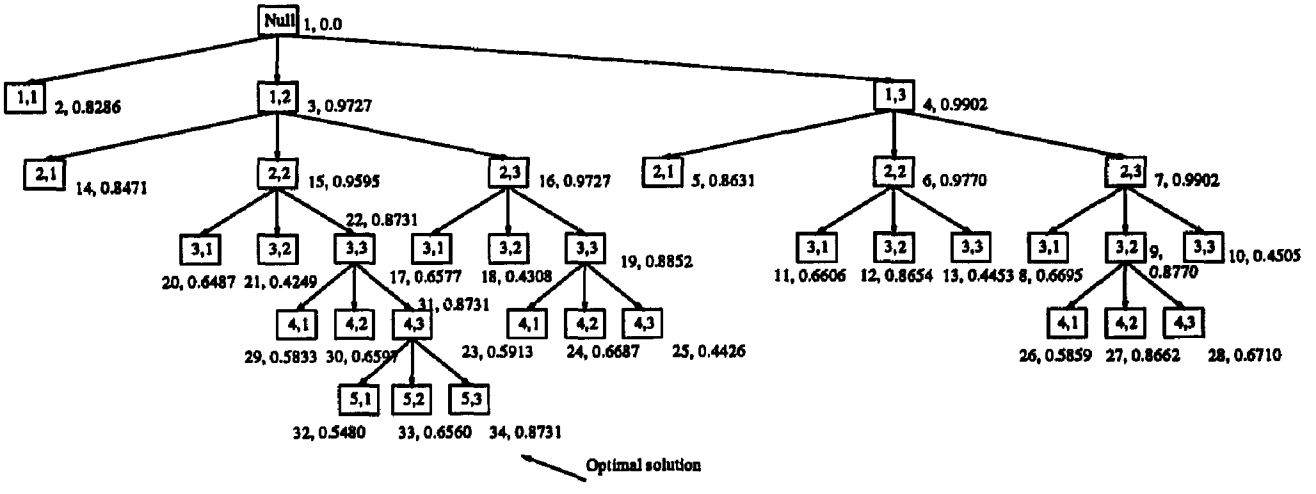
Several interesting properties observed in the experiments are given below.

P1. The **MAA** tends to allocate sequentially–executing modules subject to the same tight timing constraints to the same PN. For example, the best allocation of the TG in Fig. 2.4 to the distributed system represented by a complete graph of 3 PNs and with homogeneous node failure rates ($\lambda_k = 0.001$) and link failure rates ($\lambda_{mn} = 0.001$) is to assign T_1 to N_1, and both T_2 and T_3 to N_2. The **MAA** recognizes that the execution path $M_2 \rightarrow M_3 \rightarrow M_4 \rightarrow M_8 \rightarrow M_9 \rightarrow M_{10} \rightarrow M_{11}$ in the TG is critical subject to

[12] A TG is said to have a high-degree parallelism if most of its modules can be executed in parallel when there are enough resources. This could occur if the TG contains AND-subgraphs with a large number of branches and/or most tasks in the TG do not communicate with one another so that only a few precedence constraints are imposed on the modules belonging to different tasks.

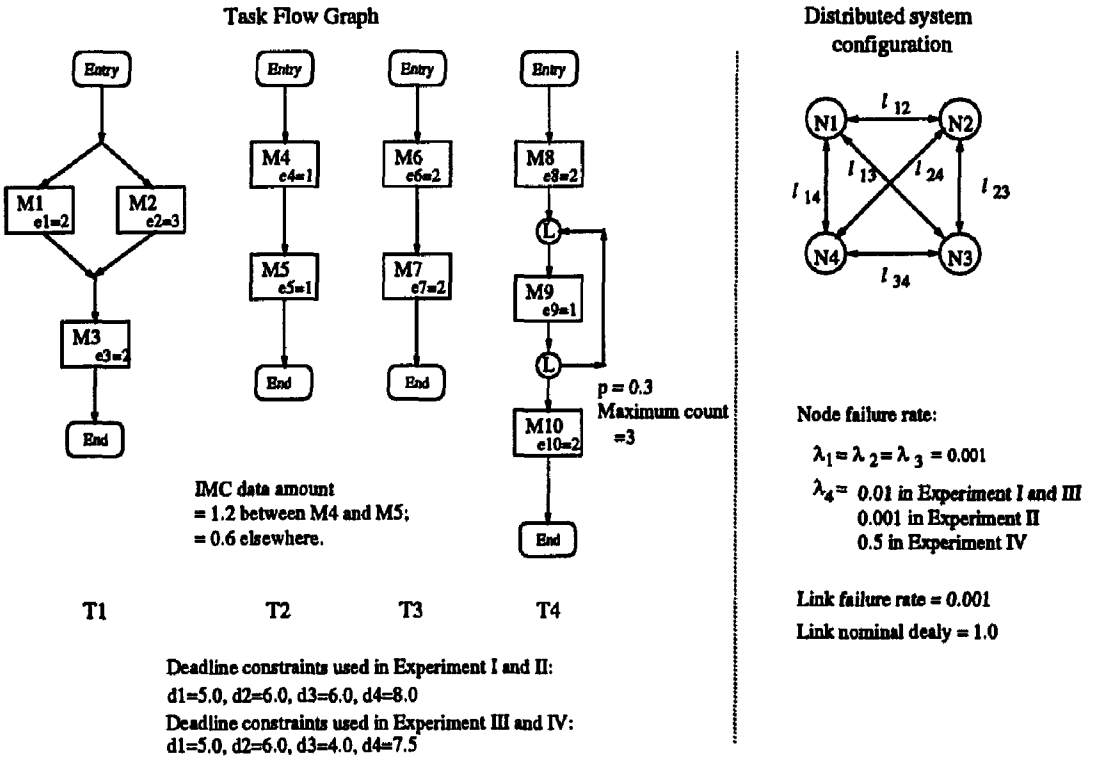


(a) The task graph and the system configuration used.

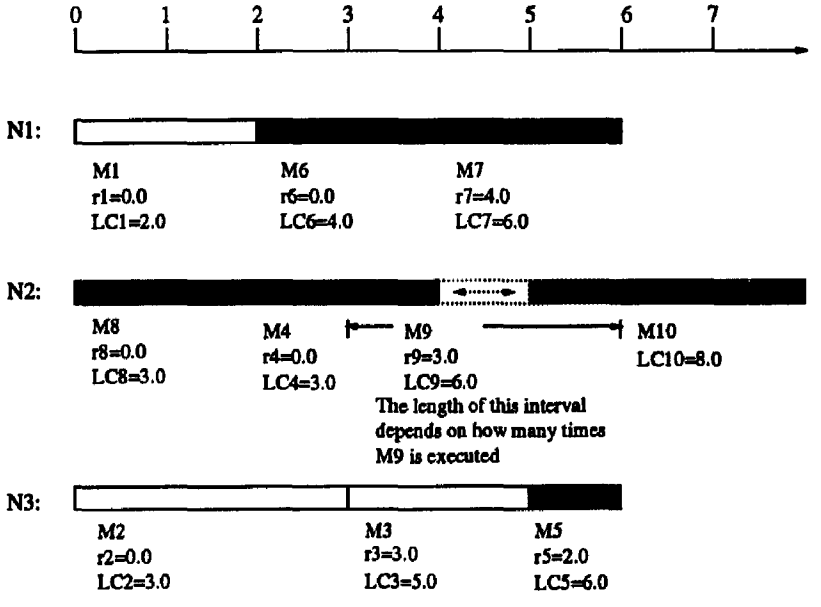


(b) The vertices visited in the state–space–tree. The numbers beside each box are the order the vertex was visited in the searching process and the corresponding upper–bound value of the objective function, respectively.

Figure 2.10: An example which shows how vertices in the state–space–search tree are visited by the **MAA**.

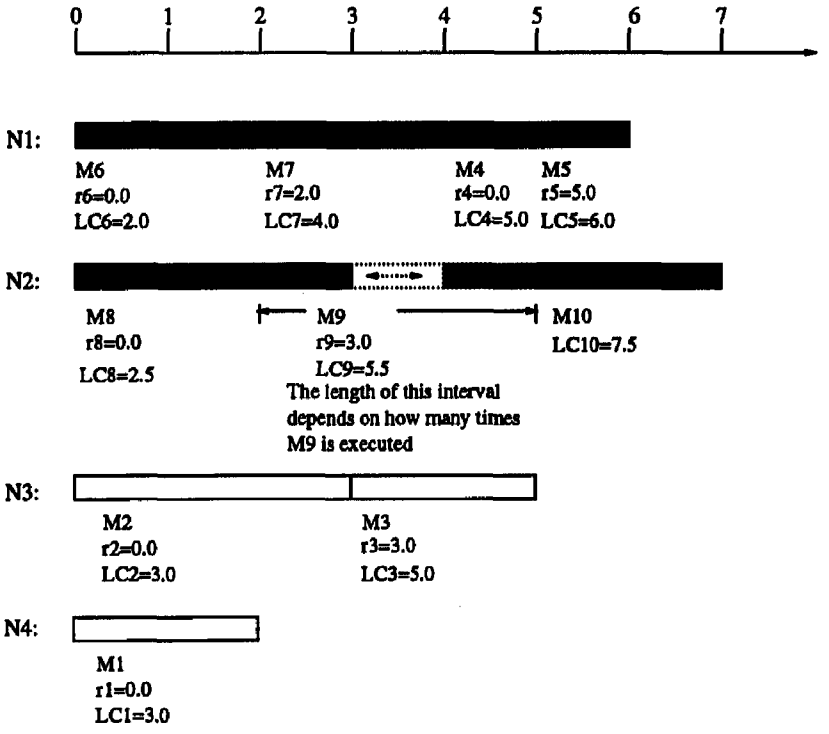


(a) The task graph and the system configuration used.

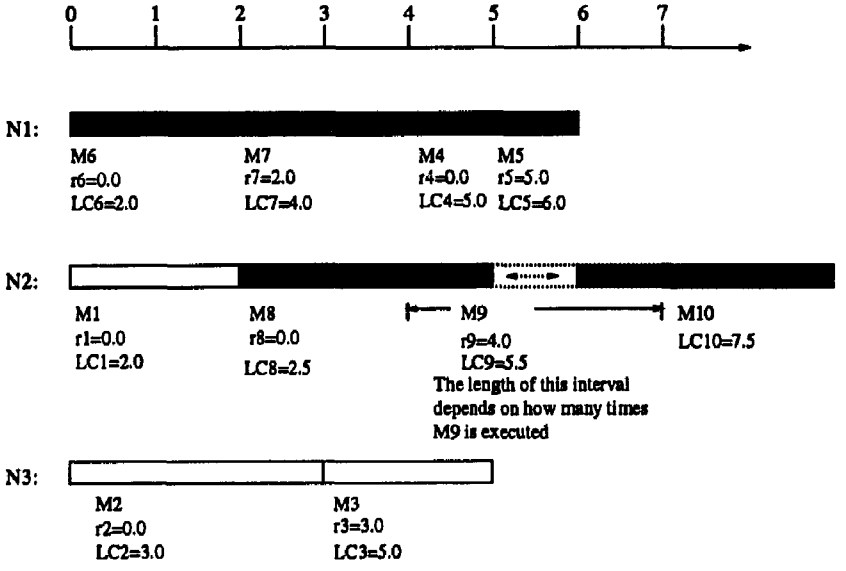


(b) Allocation and PN schedules for Experiment I: M_1, M_6, and M_7 are assigned to N_1; M_4, M_8, M_9, and M_{10} are assigned to N_2; M_2, M_3, and M_5 are assigned to N_3.

Figure 2.11: An example showing how the **MAA** allocates modules.



(c) Allocation and PN schedules for Experiment II and III: M_4, M_5, M_6, and M_7 are assigned to N_1; M_8, M_9, and M_{10} are assigned to N_2; M_2 and M_3 are assigned to N_3; and M_1 is assigned to N_4.



(d) Allocation and PN schedules for Experiment IV: M_4, M_5, M_6, and M_7 are assigned to N_1; M_1, M_8, M_9, and M_{10} are assigned to N_2; M_2 and M_3 are assigned to N_3.

Figure 2.11: (continued) An example showing how the MAA allocates modules.

T_3's deadline and cannot tolerate any IPC delay, thus allocating both T_2 and T_3 to the same PN. The resulting best objective function value is $P_{ND} = 9.8227 \times 10^{-1}$.

P2. Heavily communicating modules may not necessarily be allocated to the same PN. For example, consider the allocation of the TG in Fig. 2.11(a). The attributes of both the TG and the distributed system are specified in Experiment I. As shown in Fig. 2.11(b), the **MAA** allocates M_1, M_6 and M_7 to N_1; M_4, M_8, M_9 and M_{10} to N_2; M_2, M_3 and M_5 to N_3 so that all modules meet their latest completion times ($P_{ND1} = 1.0$) and are allocated to the most reliable PNs, N_1–N_3 ($P_{ND2} = 9.7933 \times 10^{-1}$). Although the IMC between M_4 and M_5 is twice more than the others, M_4 and M_5 are allocated to different PNs. This is mainly because T_2 has a less tight timing constraint than others and can thus allow IPCs among its modules. This observation is in sharp contrast to the common notion that heavily communicating modules should always be co-allocated [RSS90, BNG92].

P3. If the distributed system is homogeneous, the **MAA** assigns modules, subject to task timing constraints, in such a way that as few IPCs as possible will occur. To demonstrate this tendency, consider Experiment II in Fig. 2.11. The attributes of both the TG and the distributed system remain the same as in Experiment I except that $\lambda_4 = 0.001$ (i.e., the distributed system becomes homogeneous). Now, the allocation and schedules specified in Fig. 2.11(c) give the best solution ($P_{ND1} = 1.0$, $P_{ND2} = 9.8096 \times 10^{-1}$). Note that the only IPC occurs between M_1 and M_3, which cannot be eliminated because all modules of T_1 cannot be allocated to the same PN under T_1's timing constraint.

P4. If both timeliness and logical correctness cannot be achieved at the same time, the **MAA** maximizes P_{ND} by making a compromise between these two objectives. This is demonstrated by conducting three experiments: in Experiment I, the deadlines of the four tasks are set as $d_1 = 5.0$, $d_2 = 6.0$, $d_3 = 6.0$, and $d_4 = 8.0$. As shown in Fig. 2.11(b), the **MAA** allocates modules only to (three) reliable PNs while meeting the timing constraints ($P_{ND1} = 1.0$, $P_{ND2} = 9.7933 \times 10^{-1}$). As the deadline constraints get tighter in Experiment III, i.e., d_1 and d_2 remain unchanged while d_3 becomes 4.0 and d_4 becomes 7.5, the **MAA** is "forced" to allocate some of the modules (M_1) to a less reliable PN (N_4) in order to meet all timing constraints. Fig. 2.11(c) gives the best allocation and schedules: T_2 and T_3 are now allocated to N_1, T_4 to N_2, and T_1 to N_3 and N_4 ($P_{ND1} = 1.0$, $P_{ND2} = 9.6346 \times 10^{-1}$). On the other hand, if N_4 is highly prone to failure as is assumed in Experiment IV (λ_4 is increased

from 0.01 to 0.5), the **MAA** decides not to use N_4 at the risk of not making task T_4's deadline, as depicted in Fig. 2.11(d) ($P_{ND1} = 0.7$, $P_{ND2} = 9.8096 \times 10^{-1}$, and $P_{ND} = 6.8667 \times 10^{-1}$).

2.7.3 Practicality of the MAA

To test the practicality of the **MAA** for reasonably large TGs and/or distributed systems, we ran experiments on (1) TGs with 4–50 modules, while varying module execution times, IMC volumes, task deadlines, and randomly generating precedence constraints; (2) distributed system topologies with 3–40 PNs, while randomly varying link nominal delays and the degree of network connectivity. We then computed the ratio of the number of search-tree vertices visited to the total number, $K_{pn}^{N_M+1} - 1$, of vertices in the search tree. This — rather than the actual CPU run time — gives the general cost characteristics of the **MAA**. The numerical results for different combinations of N_M and K_{pn} are summarized in Table 2.2 and Fig. 2.12. The number of trials in each combination of N_M and K_{pn} was determined[13] so that a 95% (90%) confidence level may be obtained for a maximum error within 10% of the *average* numbers reported for $N_M \leq 10$ ($N_M > 10$). Also given in each combination are the worst and best results ever found in these trials.

In all the experiments conducted, no more than 9% of the search-tree vertices were visited before finding the best allocation for $N_M \geq 6$ and $K_{pn} \geq 3$. Also, the percentage of search-tree vertices visited falls drastically as N_M and/or K_{pn} grows, as shown in Table 2.2. This is because the 'increasing rate' for the number of vertices visited as N_M and/or K_{pn} grows is far lower than exponential, as depicted in Fig. 2.12. This suggests that both the dominance relation and the UBOF derived effectively prune unnecessary search paths at early stages of the **BB** process.

According to our experimental experiences, however, it takes a significant amount of CPU run time (usually over 10 hours on a SPARC station) for a single experiment for $N_M \geq 40$ and $K_{pn} \geq 30$, which makes collecting statistics difficult (although obtaining the best allocation for a single experiment is still computationally tractable). Some new techniques might be needed to reduce the search space. For example, based on the observation **P1**, one can co-allocate sequentially–executing modules in the TG subject to the same tight timing constraints. This can be done by calculating the release times and the latest completion times of all modules, while ignoring all IPC delays. If some module, M_i, has

[13] Under the assumption that the parameter to be estimated (i.e., the mean number of search–tree vertices visited) has a normal distribution with unknown mean and variance.

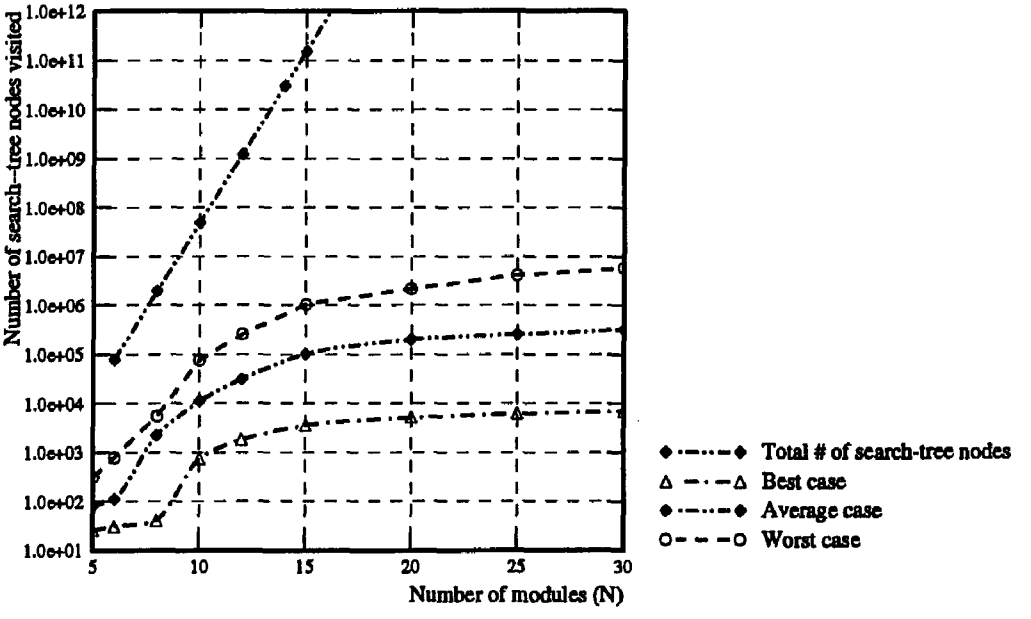
N_M		6	8	10	15	20	30	40
Best case	# visited	19	25	166	649	1488	2432	3576
	% visited	0.87%	0.13%	0.09%	0.002%	—	—	—
Average case	# visited	47	442	2230	12720	37635	56015	68687
	% visited	2.15%	2.25%	1.26%	0.03%	3.60×10^{-4}%	—	—
Worst case	# visited	159	1240	14680	79905	150304	268420	374572
	% visited	7.27%	6.30%	8.29%	0.19%	1.44×10^{-3}%	—	—
$K_{pn}^{N_M+1} - 1$		2186	19682	177146	4.305×10^7	1.046×10^{10}	6.177×10^{14}	3.647×10^{19}

(a) $K_{pn} = 3$

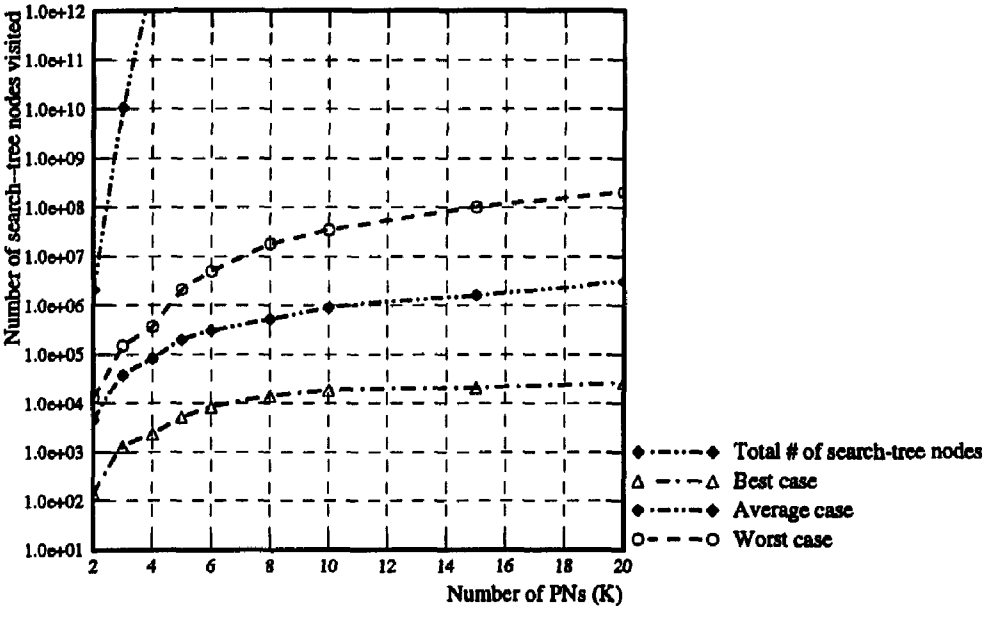
K_{pn}		2	3	4	5	6	8	10
Best case	# visited	21	166	233	426	581	705	814
	% visited	1.03%	0.09%	5.56×10^{-3}%	8.72×10^{-4}%	1.60×10^{-4}%	—	—
Average case	# visited	360	2230	7452	9388	11068	17038	27026
	% visited	17.59%	1.26%	0.178%	0.019%	3.05×10^{-3}%	1.98×10^{-3}%	—
Worst case	# visited	723	14680	35335	50353	56671	63248	83035
	% visited	35.32%	8.29%	0.842%	0.103%	0.016%	7.36×10^{-4}%	—
$K_{pn}^{N_M+1} - 1$		2047	177146	4.194×10^6	4.883×10^7	3.628×10^8	8.590×10^9	1.00×10^{11}

(b) $N_M = 10$

Table 2.2: The number and percentage of vertices visited in the search tree by **MAA**. — indicates less than $10^{-6} \times 100\%$ of nodes in the search-tree were visited.



(a) $K_{pn} = 5$



(b) $N_M = 20$

Figure 2.12: Number of search-tree vertices visited by **MAA**

$LC_i - r_i$ equal to e_i[14] in all the component graphs of the TG, then this module and both its preceding and succeeding modules which are subject to the same timing constraint should be co-allocated.

2.8 Conclusion

We have addressed the problem of allocating periodic task modules in a distributed real-time system subject to precedence constraints, timing requirements, and intermodule communications. The probability of no dynamic failure is used as the objective function to incorporate both the timeliness and logical correctness of real-time tasks/modules into module allocation. The **MAA** not only assigns modules to PNs, but also uses the **MSA** to schedule all modules assigned to each PN.

An interesting finding from the numerical experiments is that the common notion in general-purpose distributed systems that heavily communicating modules should be co-allocated [BNG92, RSS90] may not always be applicable to real-time systems. Based on a set of experiments using randomly-generated TGs and distributed systems, the **MAA** has also been shown to be computationally tractable for $N_M \leq 50$ and $K_{pn} \leq 40$. The percentage of search tree vertices visited is also shown to fall drastically as N_M and/or K_{pn} grows, suggesting that both the dominance relation and the UBOF derived here provide an effective means of limiting the growth of the search tree without removing those paths leading to optimal solutions.

Despite its advantages mentioned above, the **MAA** still takes a significant amount of time to locate an optimal allocation for the case of $N_M \geq 40$ and $K_{pn} \geq 30$ due to the fact that there exist an extremely large number of search paths which might lead to an optimal solution and thus cannot be pruned at early stages of the BB process. One challenging extension to this research is to investigate the problem of grouping modules and/or PNs to reduce the size of the search space without resorting to a heuristic-directed technique. The conditions under which modules could be co-allocated, e.g., **P1** and **P2** observed in Section 2.7, should be explored further.

[14] or, $LC_i - r_i \leq f_m \cdot e_i$, where $f_m \geq 1.0$ is empirically determined if suboptimal allocations are allowed.

CHAPTER 3

LS USING BAYESIAN DECISION THEORY

3.1 Introduction

As discussed in Chapter 1, each node gathers state information of other nodes using the information policy, and makes the transfer/location decision based on the state information gathered. No matter which information policy is used, the state information gathered may be out–of–date due to the communication delay incurred in state information collection and task transfer [CK87]. That is, a node's *observed* states of other nodes may be different from their *true* states at the time of making LS decisions. This inconsistency often causes a node to transfer an overflow task to an actually incapable node, and degrades the performance of adaptive LS. The performance degradation caused by communication delays, despite its importance, is seldom addressed in literature (except for [MTS89b, MTS89a]). In this chapter, We propose a new LS scheme using Bayesian decision theory as well as the concept of buddy sets, preferred lists, and state–change broadcasts in [SC89a] to reduce the performance degradation caused by communication delays. The basic ideas used are detailed in Section 3.2. The Bayesian decision model used is presented in Section 3.3. How both the components of the Bayesian decision model and the concepts presented in [SC89a] can be accommodated into our LS scheme is also described there. Section 3.4 describes how each node constructs prior and posterior probability distributions, and updates loss–minimizing decisions. Using several performance metrics, such as P_{dyn}, task transfer–out ratio, and maximum system utilization, we comparatively evaluate the proposed scheme along with five other schemes: no LS; LS with state probing, focused addressing and random selection; and perfect LS. As indicated in the numerical results in Section 3.5, the proposed scheme outperforms all but the perfect LS scheme in minimizing P_{dyn}. We also study in Section 3.5 via simulation the impact of the time–varying behavior of task arrivals on the performance of the proposed scheme (in particular, on the accuracy of Bayesian analysis). This chapter concludes with Section 3.6.

3.2 Basic Ideas of the Proposed Scheme

In order to reduce the overheads associated with state collection and task transfer, the LS scheme in [SC89a] requires each node to collect and maintain the state information of only those nodes in its physical proximity, called a *buddy set*. When a node cannot complete a real-time task in time, only those nodes in its buddy set are considered for transferring this task. Moreover, these buddy sets overlap with one another so that an overflow task may be transferred from an overloaded node to some other node which is a member of a different buddy set. That is, those tasks arrived at a congested region — in which most nodes cannot complete all of their own tasks in time — can be shared by the entire system, rather than overloading the nodes in the region.

Based on the topological property of the system, each node orders all the other nodes according to the distance from itself into its *preferred list*. Each preferred list should be arranged so that **(P1)** every node in the system is selected as the kth preferred node of one and only one other node, $\forall k$; **(P2)** if node i is the kth preferred node of node j, then node j is the kth preferred node of node i. If there are multiple nodes of the same distance, they are ordered based on their location, e.g., east, south, west, and north in case of square meshes. Once each node's preferred list is constructed, the node's buddy set can be formed with any required number of nodes counting from the top of its preferred list. When a node is unable to complete a task in time, it will transfer the task to the first 'capable node' found in its preferred list. As will be rigorously proved in Chapter 4, with the properties of **(P1)**-**(P2)**, the preferred list provides a means to select a receiver node among several possible candidate nodes while minimizing the probability of more than one overloaded node simultaneously sending tasks to the same underloaded node.

In [SC89a] four state regions determined by three thresholds of QL are used to characterize the workload of each node: underloaded, medium-loaded, fully-loaded, and overloaded. A node will broadcast the change of state region to the nodes in its buddy set only when it switches from underloaded to fully-loaded and vice versa. The state information kept at each node is thus up-to-date as long as the broadcast delay is not significant. Communication delays may still occur and thus degrade system performance unless the size of buddy set is kept very small, in which case the LS capability of the whole system may not be fully utilized. Thus, Bayesian decision theory is used to counter the communication delay problem as shown in Fig. 3.1.

Fig. 3.1 shows the actions that the scheduler on each node should take for the following four cases:

At each node:
When a task T_i with execution time e_i and laxity ℓ_i arrives at the node:
determine the position, j_p, in the task queue Q†such that $\ell_{j_p-1} \leq \ell_i \leq \ell_{j_p}$;
if current_time + $\sum_{k=1}^{j_p-1} e_k \geq \ell_i$ **then**
begin
receiver_node := table_lookup($\underline{O}$:observation, ℓ_i:laxity)‡;
transfer task T_i to receiver_node;
end
else
begin
queue task T_i in position j_p;
for $k = j_p + 1, length(Q)$
begin
if current_time + $\sum_{j=1}^{k-1} e_j \geq \ell_k$ **then**
begin
receiver_node := table_lookup($\underline{O}$:observation, ℓ_k:laxity);
dequeue and transfer T_k to receiver_node;
end
end
if current_CET crosses TH_{2k}, $1 \leq k \leq \lceil \frac{K_i}{2} \rceil - 1$, **then**
/* $TH_1, \cdots, TH_{K_i-1}$ are thresholds */
broadcast the state–region change to all nodes in its buddy set;
end

When a broadcast message arrives from node i, $1 \leq i \leq n$:
update observation of node i's state, O_i;
record the (observation, true state) pair needed for constructing probability distributions;

When current_CET crosses TH_{2k}, $1 \leq k \leq \lceil \frac{K_i}{2} \rceil - 1$:
broadcast the state–region change to all nodes in its buddy set;

At every T_p clock ticks:
update the probability distributions and the table of loss–minimizing decisions;

†The task queue Q is ordered by task laxities.
‡If a node anticipates, based on the current observation $\underline{O}$, that no other node can complete the task in time, this task is declared to be lost and discarded.

Figure 3.1: Operation of the task scheduler on each node.

(a) When a new task arrives,

(b) When a state–region change broadcast is received,

(c) When current CET crosses TH_{2k}, $1 \leq k \leq \lceil \frac{K_t}{2} \rceil - 1$, and

(d) At every T_p clock ticks.

Those tasks already queued at a node are sorted by their laxities and executed on a minimum–laxity–first–served (MLFS) basis. Upon arrival of a real–time task at a node, the scheduler checks if the CET on that node contributed by those tasks with laxity smaller than this task is less than, or equal to, the laxity of the new task. If it is not, the new task has to be transferred, and the node's task queue remains unchanged; if it is, the new task is inserted in the task queue, and if this insertion leads to violation of existing guarantees,[1] those tasks whose guarantees are violated need to be transferred to other capable nodes.

K_t state regions obtained from $K_t - 1$ thresholds, TH_1, TH_2,...,TH_{K_t-1}, are used to describe the workload of each node.[2] Each node will broadcast a time–stamped message, informing all the other nodes in its buddy set of a state–region change whenever its load crosses TH_{2k} for some k, where $1 \leq k \leq \lceil \frac{K_t}{2} \rceil - 1$. The reason for not broadcasting the change of state region whenever a node's load crosses any threshold is to reduce the network traffic resulting from region–change broadcasts. Moreover, the reason for not combining two adjacent state regions into one and then broadcasting the change of state region whenever a node's load crosses any threshold is to include finer information in each broadcast and thus construct more accurate posterior distributions.

By collecting time–stamped state samples and by keeping track of the corresponding observations at the times these samples were taken, each node can construct the prior/posterior distributions. These distributions characterize the inconsistency between the node's observed and true states of other nodes, and are used to periodically (once every T_p clock ticks) update the loss–minimizing decisions with Bayesian decision theory. As will become clear, the undesirable effects of the delay in broadcasting state–region changes/transferring tasks are eliminated by using these prior/posterior distributions. Whenever a node cannot complete a task in time, the node's scheduler looks up the list of loss–minimizing decisions, and choose — based on the current state information — the best candidate node for transferring this task such that the expected loss is minimized

---

[1] By 'guarantee', we mean the node has enough resources to complete the task in time upon its arrival. A granted guarantee may be deprived later because of the arrivals of tighter–laxity tasks under the MLFS discipline.

[2] K_t is a tunable parameter which will be discussed later.

with respect to the conditional (or posterior) probability distribution.

3.3 Bayesian Decision Model

Conceptually, the task scheduler of a node can be modeled as a Bayesian decision maker. In what follows, we shall describe how the proposed LS scheme can be cast into a Bayesian decision model.

3.3.1 Preliminaries

The Bayesian statistical structure is a very powerful modeling tool when one has to make decisions based on some observations. The elements of a Bayesian decision problem are a parameter space (a space of state of nature) Ω, a decision space D, and a real-valued loss function L which is defined on the product space $\Omega \times D$ [Ber86, DeG70]. For any point $(\omega, d) \in \Omega \times D$, the quantity $L(\omega, d)$ represents the loss when the value of the outcome W of the space Ω is ω and d is the decision chosen.

If P is any given probability distribution of the parameter W, then for any decision $d \in D$, the expected loss or risk, $\zeta(P, d)$, is given by

$$\zeta(P, d) = \int_{\Omega} L(\omega, d)\, dP(\omega). \tag{3.1}$$

It will be assumed that the integral in Eq. (3.1) is finite for every $d \in D$.[3] We now want to choose a decision d which minimizes the risk $\zeta(P, d)$. The *Bayes risk* $\zeta^*(P)$ is thus defined to be the greatest lower bound for the risks $\zeta(P, d)$, $\forall d \in D$, i.e.,

$$\zeta^*(P) = \inf_{D} \zeta(P, d). \tag{3.2}$$

Any decision d^* whose risk is equal to the Bayes risk is called a *Bayes decision* with respect to the distribution P.

In many decision problems like the one we are going to discuss, before choosing a decision from D, we observe the value of a random variable O that is related to the parameter W. The observation of O provides us with some information about the value of W which may be useful in making a good decision. The essential component of problems of this kind is, in addition to a parameter space Ω, a decision space D, and a loss function L, a family of sampling functions $\{P_{O|W}(. \mid \omega), \omega \in \Omega\}$ of observation O. Let S denote the sample space of all possible values of O. With the family of sampling functions and the

[3] Any decision d for which this assumption is not true can usually be eliminated from the set D.

(prior) probability distribution, P, of W, we can calculate the conditional distribution of W given O, $P_{W|O}$, as:

$$P_{W|O=o}(\omega) = \frac{P_W(\omega)P_{O|W}(o \mid \omega)}{\int_\Omega P_W(\omega)P_{O|W}(o \mid \omega)d\omega}. \tag{3.3}$$

Now, we must choose a decision function δ which specifies, for every possible value $o \in S$, a decision $\delta(o) \in D$ with the expected loss given the observation o as:

$$\zeta(P_{W|O=o}, \delta(o)) = \int_\Omega L(\omega, \delta(o))\, dP_{W|O=o}(\omega). \tag{3.4}$$

Note that Eq. (3.4) is almost the same as Eq. (3.1) except that P has been replaced by $P_{W|O=o}$. That is, given the observation of O, the decision problem remains unchanged except that the distribution of W has changed from the prior to the posterior distribution. Thus, any minimizing decision $\delta^*(o)$ is simply a decision which yields the smallest expected loss under the conditional distribution of W when the observed value of O is o. In other words, $\delta^*(o)$ is a Bayes decision against the conditional distribution of W when $O = o$.

3.3.2 Components of the Bayesian Decision Model

This subsection describes how to apply Bayesian decision theory to adaptive LS, and how to accommodate both the components of the statistical model and the concept of [SC89a] into our scheme.

Parameter Space

The parameter space is defined as $\Omega = \Omega_1 \times \Omega_2 \times ... \times \Omega_{N_B}$, where N_B is the number of nodes in a buddy set, and Ω_i is the parameter space for node i. Note that the size of state space and the overhead of broadcasting state–region changes are greatly reduced by using buddy sets. The parameter space, Ω_i, may be defined by QL, CET, resource available time (RAT) on node i, or a combination thereof, depending on task characteristics and performance requirements. For example, if all tasks have an identical execution time, QL suffices to express each node's workload; otherwise, CET must be used. Since in a real–time system, execution time varies from task to task, the state of a node is defined to be its CET. The dimension of the state can be augmented if more than one resource are needed to execute tasks.

Probability Distribution on Parameter Space

The probability distribution on parameter space is the joint probability distribution of Ω_i's, e.g., $P_W(\underline{\omega}) = P_W(\omega_1, \omega_2, ..., \omega_{N_B})$, where ω_i is the CET of node i and N_B is

the number of nodes in a buddy set. The marginal probability distribution on Ω_i, P_{W_i}, can be obtained from P_W by integration. We construct these probability distributions by collecting state samples through region–change broadcasts (to be discussed in Section 3.4).

Set of Available Decisions

The set of available decisions is $D = \{d_1, d_2, ..., d_n\}$, where d_i denotes the decision to move one task from the current node to node i. Other options are also possible. For example, if a locally overflow task is extremely important, then one may want to move it simultaneously to two or more nodes so that the probability of dynamic failure can be minimized. In such a case, a decision d_{ij} is added to the set of available decisions, which denotes the transfer of a task to both node i and node j.

Set of Loss Functions

The set of loss functions is defined as $\{L^{T_d}, T_d \in (0, L_{max}]\}$, where L describes the 'loss' resulting from each combination of state and decision, given that the laxity — which equals deadline – execution time – current time — of a locally overflow task is T_d. L_{max} is the largest task laxity in the system. If P_{dyn} is the main concern, the loss function may be defined as:

$$L^{T_d}(\underline{\omega}, d_i) = \delta(\omega_i - T_d) = \begin{cases} 1, & \text{if } \omega_i - T_d \geq 0, \\ 0, & \text{otherwise,} \end{cases}$$

where $\delta(x)$ is the unit step function. In such a case, minimizing the expected loss is equivalent to minimizing the probability of dynamic failure. The loss function can also be defined as

$$L^{T_d}(\underline{\omega}, d_i) = \omega_i - T_d,$$

if the task needs to be executed not only before its deadline but also as early as possible.[4]

Sample Space of Observation

The sample space of observation, S, is the set of all possible observations. Specifically, $S = S_1 \times S_2 \times \cdots \times S_{N_B}$, where S_i is obtained by dividing the parameter space W_i for each node i into the K_i regions determined by $K_i - 1$ thresholds, TH_1, TH_2,..., TH_{K_i-1}. Node i is said to be in the k-th region if $TH_k \leq \omega_i < TH_{k+1}$, where $k \geq 0$, and $TH_0 \triangleq 0$.

Note that the knowledge of a node's state region is not sufficient to determine accurately its capability of completing arbitrary tasks in time. For example, a node with

[4] L^{T_d}, could be negative in this case; the more negative L^{T_d}, the more early the task is executed.

its state in a high–numbered state region may still be able to complete an arrived task with a large laxity in time, whereas a task with a small laxity may not be completed in time even by a first–region node if the CET on that node is greater than the task's laxity. Thus, unlike in [SC89a, PTS88, WM85, ELZ86], these thresholds only serve as reference points, rather than indicating a node's capability of meeting task deadlines. As will be discussed in Section 3.5, the performance of the proposed scheme is rather insensitive to the choice of threshold values.

Each node will broadcast a time–stamped message, informing all the other nodes in its buddy set of a state–region change whenever its state crosses TH_{2k}, $1 \leq k \leq \lceil \frac{K_i}{2} \rceil - 1$. Upon receipt of a region–change broadcast, every node in the buddy set will update its observation of the broadcasting node accordingly. The delay in broadcasting a region–change may cause inconsistency between the observed and true states of a node. We will characterize this inconsistency by constructing prior/posterior distributions (to be discussed in Section 3.4). So, based on the observation o_i, a node can estimate the state of node i by using the prior/posterior distributions constructed from the samples collected through time–stamped region–change broadcasts.

Family of Sampling Functions

The family of sampling functions, $\{P_{O|W}(. \mid \underline{\omega}), \underline{\omega} \in \Omega\}$, describes the conditional probability distribution of the observation O given the state $W = \underline{\omega}$. These probability distributions are derived from the samples gathered through time–stamped region–change broadcasts. With the prior probability distribution P_W and these sampling functions $P_{O|W}$, one can derive the posterior probability distribution $P_{W|O}$ by using the Bayes rules [Ber86] that is needed to compute the expected loss with observations.

3.4 Region–Change Broadcasting, Prior/Posterior Probability Distributions and Loss–Minimizing Decisions

As mentioned earlier, the delay in region–change broadcasts may cause the collected information to be out–of–date. For example, consider the following scenario: after broadcasting a state–region change, say from 3 to 1, node i switches back to region 3 due to the arrival of new tasks and/or transferred–in tasks.[5] Upon receipt of the broadcast from node i, node j may decide to send a task to node i, since it is unaware that node i

[5]These tasks may have been sent by other nodes before the broadcast, but arrived at node i after the broadcast due to task–transfer delay.

has switched back to region 3 shortly after broadcasting the 3→1 region–change. If node j, instead of hastily believing in what it observed, can compute the probability that node i is indeed capable of completing task(s) in time and decide whether or not to send the task to node i, then the probability of dynamic failure could be significantly reduced. To this end, we shall characterize the inconsistency between the observed and true states with prior/posterior distributions.

The first step is to construct both the probability distribution on the parameter space and the conditional probability distribution of an observation. These two distributions, in general, vary over both nodes and time in a dynamic environment. Thus, to monitor the dynamics of the system, each node must collect state samples on-line and construct these distributions from the samples gathered via region–change broadcasts. The methods for collecting state samples, constructing probability distributions, and deriving loss–minimizing decisions are discussed in the following subsections.

3.4.1 Collection of State Samples

Whenever a node's state crosses TH_{2k} $(1 \leq k \leq \lceil \frac{K_4}{2} \rceil - 1)$, the node will broadcast, to all the other nodes in its buddy set, a time-stamped message which contains node number i, the state ω_i^b before the change of state region, the state ω_i^a after the change, and the time t_0 when ω_i^b was sampled. When the message broadcast by node i arrives at node j, node i's state ω_i^b can be recovered by node j using the node number field and the state field, from which P_W can then be calculated. Node j can also trace back to find out its observation o_i at time t_0. This observation o_i is node j's observation of node i's state at the time when node i was actually in state ω_i^b. o_i's along with ω_i^b's are used to construct $P_{O|W}$. (Here we assume that the node clocks are synchronized to establish a global time–base. A scheme for achieving this synchronization is presented in [RKS90].) Any inconsistency between ω_i^b and o_i at time t_0 is characterized by this probability distribution. The only effect of the delays in task transfers and region–change broadcasts is that messages (tasks) may not arrive at a node immediately after their broadcast (send), and thus, may become obsolete upon their arrival at other nodes. The correctness of all samples gathered is, however, not affected by these delays. Besides, ω_i^a sent by node i at time t_0 is considered as node j's new observation of node i at the time this message is received, rather than at time $t_0^+ > t_0$.

A primary advantage of region–change broadcasts over periodic state broadcasts is the elimination of the need to determine an "optimal" exchange period — a very difficult problem since it depends on workload characteristics, and has to weigh the tradeoff between the resulting increase in network traffic and the negative effect of using out–of–date infor-

mation. Moreover, as we shall see, the threshold values have only minor effects on system performance (as a result of using Bayesian analysis).

3.4.2 Derivation of Probability Distributions

Each node updates, once every T_p units of time, the probability distributions using all the samples gathered so far, and re-calculates the loss–minimizing decisions. T_p should be chosen to reflect the fluctuation of system load and the number of samples required for the specified level of confidence in the results obtained.

The general rule for updating the probability distribution of W is

$$P_U = aP_T + (1-a)P_O$$

where P_U is the updated probability distribution, P_T is calculated from the samples gathered over the last T_p units of time, and P_O is the old probability distribution. That is, the updated probability distribution is a weighted sum of the distribution calculated from the samples gathered within the last T_p units of time and the old probability distribution. The ratio a ($0 < a \leq 1$) represents the tradeoff between obtaining better averages and reflecting load changes. One may increase (decrease) a if system load varies rapidly (slowly). The same rule may be applied to update the sampling functions, $P_{O|W}$.

Non-informative probability distributions (e.g., uniform distributions) or some default probability distributions (obtained from previous experiences) may be used as the initial distribution of W and the sampling functions. According to our simulation results, the performance of the proposed scheme is found to be rather insensitive to the choice of an initial probability distribution. Each node may initially rely on the preferred list for LS decisions. This is because both prior and posterior distributions will be iteratively updated as time goes on, and usually represents the true system characteristics after two or three updates. Besides, if the task arrival pattern on each node does not change drastically with time, the probability updating process need not be executed often once the probability distributions are well–tuned.

3.4.3 Calculation of Loss–Minimizing Decisions

With the prior distribution of W and the sampling function, $P_{O|W}$, one can calculate the posterior distribution $P_{W|O}$ using the Bayes rule (Eq. (3.3)). For each possible observation $\underline{o} \in S$ and for each possible laxity $T_d \in (0, L_{max}]$, a node then computes the

expected loss associated with the decision d_i given the observation $\underline{o}$ and the laxity T_d as:

$$\zeta^{T_d}(P_{W|O=\underline{o}}, d_i) = \int_{\Omega} L^{T_d}(\underline{\omega}, d_i) dP_{W|O}(\underline{\omega}) \tag{3.5}$$

for $i = 1, ..., N_B$. The decision $d_i = \delta^{T_d}(\underline{o})$ that yields the minimum expected loss is chosen as the optimal decision given the observation $\underline{o}$. A tie will be broken by choosing, from the preferred list, the first d_i with the minimum expected loss.[6] Because of the way $L^{T_d}(\underline{\omega}, d_i)$ was defined and the assumption that W_i is stochastically independent of the state of node j for $j \neq i$ [ELZ86], the computation of the expected risk, $\zeta^{T_d}(P_{W|O=\underline{o}}, d_i)$, depends only on the marginal probability distribution, $P_{W_i|O_i}(\omega_i)$. That is, if $L^{T_d}(\underline{\omega}, d_i) = \delta(\omega_i - T_d)$, then

$$\begin{aligned}
\zeta^{T_d}(P_{W|O=\underline{o}}, d_i) &= \int_{\Omega} \delta(\omega_i - T_d)\, p_{W|O=\underline{o}}(\underline{\omega}) d\underline{\omega} \\
&= \int_{\Omega_i} \delta(\omega_i - T_d)\, p_{W_i|O=\underline{o}}(\omega_i) d\omega_i \\
&= \int_{\Omega_i} \delta(\omega_i - T_d)\, p_{W_i|O_i=o_i}(\omega_i) d\omega_i \\
&= P_{W_i|O_i}(W_i > T_d \mid O_i = o_i).
\end{aligned} \tag{3.6}$$

In other words, the expected loss of adopting decision d_i given the observation $\underline{o}$ and the task laxity T_d is the probability that node i's CET is greater than T_d. The second equality in Eq. (3.6) follows from the property of total probabilities, and the third equality results from the assumption that W_i is stochastically independent of the observation O_j of node j, $j \neq i$, i.e., $p_{W_i|(O_1,O_2,...,O_n)}(\omega_i) = p_{W_i|O_i}(\omega_i)$. Similarly,

$$\zeta^{T_d}(P_{W|O=\underline{o}}, d_i) = \int_{\Omega_i} (\omega_i - T_d) \cdot p_{W_i|O_i=o_i}(\omega_i)\, d\omega_i,$$

if $L^{T_d}(\underline{\omega}, d_i) = \omega_i - T_d$. The set of loss-minimizing decisions, $\{\delta^{T_d}(\underline{o}) : \underline{o} \in S, T_d \in (0, L_{max}]\}$, is a list of decisions to choose for each possible observation and each possible task laxity. Once these calculations are completed, a task scheduler only needs to look up a table when determining a LS decision for a given observation $\underline{o}$.

3.5 Performance Evaluation

To demonstrate the effectiveness of the proposed LS scheme, we simulated it under the assumption that inter-arrival times of external tasks are exponentially distributed. Note that periodic tasks constitute the "base load" of a real-time system and are usually *statically* allocated to the nodes in the system as discussed in Chapter 1. A LS scheme is then used

[6] The nice property (of the preferred lists) in distributing overflow tasks among capable nodes is thus maintained in the proposed scheme.

to dynamically distribute aperiodic tasks as they arrive, and their arrival is known to be well modeled as a Poisson process. After their initial allocation, periodic tasks, albeit rare, may be redistributed subject to aperiodic task arrivals and the current system-wide state. In such a case, a Poisson task arrival model may prove appropriate for assessing the performance of the proposed LS scheme. Note, however, that the proposed LS scheme is not restricted to Poisson models.

3.5.1 LS Schemes/Parameters Considered

The proposed scheme and three other LS schemes are comparatively evaluated. The schemes under consideration differ in the way a node treats locally overflow tasks as follows:

The state probing scheme: a node with an overflow task randomly probes up to some predetermined number of nodes and transfers the task to the first capable node found during the probing.

The random selection scheme each locally overflow task is sent to a randomly selected node.

The focused addressing scheme: each node exchanges state information periodically. A node sends its overflow task to a node (called the *focused* node) which is randomly selected among those nodes 'seen' to be capable of completing the task in time. (If such a capable node does not exist, the node itself becomes the focused node.) Meanwhile, the node also sends request–for–bid (RFB) messages to all the other nodes in the system, indicating that bids (which contain the CET of the bidding node) should be returned to the designated focused node. If the focused node cannot complete the task in time, it chooses, based on the bids received, a capable node for transferring the task (ties are broken randomly); otherwise, the task is queued on the focused node, and the received bids are used to locate the receiver nodes for those tasks, if any, whose existing guarantees become invalid as a result of accepting the transferred task. The bids received at the focused node are also used to update the observation of other nodes' states. If neither the focused node nor the bidding nodes can complete the task in time, the task is declared to be lost and thrown out. To avoid poor CPU utilization, RFB messages do not require nodes to reserve CPU cycles or any other resources needed to execute the task to be transferred until it actually arrives. When a task arrives at a node whose bid has already been accepted, the node will check

again whether or not the task can be completed in time. This is a simplified version of the scheme proposed in [RSZ89, SRC85]. It also differs slightly from that of [RSZ89, SRC85] in the way a node chooses the focused node. The authors of [RSZ89, SRC85] used the percentage of free time during the next window (which is a design parameter) and many other estimated parameters to determine the focused node or the node to which the task must be transferred again. However, we use the observed CET of other nodes to determine the node(s) for transferring tasks.

The proposed scheme: a node sends each overflow task to another node in its buddy set based on a technique that combines preferred lists, state–region change broadcasts, and Bayesian analysis.

These schemes are compared with one another as well as with two other baseline schemes: the non–cooperative scheme (where each node does not have LS capability) and the quasi–perfect LS scheme (where each node has complete information on the workload of other nodes without any overheads in collecting it.[7])

The performance of LS schemes depends on a large number of parameters which are classified into the following three groups:

(1). System parameters, such as the number, N, of nodes in the system, the degree of system heterogeneity, and the communication delay which consists of task–transfer and medium–queueing delays. The former delay depends on the capacity of the communication network and the size of the transferred task, while the latter dynamically changes with the system load.

(2). Characteristic parameters of the task set, such as the external task arrival rate, λ_i^{ext}, the laxity distribution of external tasks, and the distribution of execution time required by external tasks, on each node. For all results presented below, we use $\{e_1, e_2, ..., e_k\}_{\{q_{e_1}, q_{e_2}, ..., q_{e_k}\}}$ to denote the task set in which an external task requires execution time e_i with probability q_{e_i}, $\forall i$. If $q_{e_i} = q \ \forall e_i$, then $\{q_{e_1}, q_{e_2}, ..., q_{e_k}\}$ is condensed to q. Similarly, $\{\ell_1, \ell_2, ..., \ell_n\}_{\{q_{\ell_1}, q_{\ell_2}, ..., q_{\ell_n}\}}$ is used to describe the laxity distribution of external tasks.

(3). Design parameters of the proposed LS mechanism, such as the size of buddy sets, N_B, the number, K_i, and values of thresholds, TH_1,..., TH_{K_i-1}, and the posterior probability update interval T_p.

---

[7]This, however, cannot be modeled as an $M/G/n$ queue, as compared to the perfect load sharing in [ELZ86]. This is because of the transfer policy used which incorporates the consideration of task laxities/deadlines into the LS decision. Hence, we label this scheme as quasi–perfect.

A 16–node regular system[8] is used as an example for the simulations. For convenience, all time–related parameters are expressed in *units of average task execution time*, $E(R)$. The size of buddy sets, N_B, is chosen to be 10, since the performance improvement by increasing it beyond 10 was shown in [SC89a] to be insignificant. The maximum number of nodes to be probed randomly for each overflow task is restricted to 5 based on the finding in [ELZ86]. The computational overhead for each bidding, state probing, region–change broadcast, and probability distribution update is assumed to be 1, 1, 1, and 2 % of $E(R)$, respectively.

Each communication medium/link is equipped with buffers, and transferred tasks or broadcast messages are queued and/or transmitted in order of their arrival. No priority mechanism regulates the transmission over the medium, i.e., a FCFS rule is assumed. Unless specified otherwise, the delay associated with each task transfer is assumed to be 10 % of the execution time of the task being transferred. The queueing delay due to task transfers, region–change broadcasts, requests and responses for bids, and state probes dynamically changes with system load and traffic, and is modeled as a linear function of both (1) the average external task arrival rate and (2) the number of tasks/messages queued in the particular medium/link. (The linear coefficients are denoted as queueing delay coefficients.)

The simulation was carried out for a task set with the external task arrival rate on each node varying from 0.2 to 0.9, the ratio of $\frac{e_{j+1}}{e_j}$ $(1 \leq j \leq k-1)$ varying from 2 to 10 units of average execution time,[9] and the ratio of of $\frac{\ell_{j+1}}{\ell_j}$ $(1 \leq j \leq n-1)$ varying from 2 to 6. Due to space limitation, we present only representative results. In spite of a large number of possible combinations of task arrival rates, task execution time distributions, and task laxity distributions, the results are found to be quite robust in the sense that the conclusion drawn from the performance curves for a task set with the given task execution and laxity distributions is valid over a wide range of combinations of task execution time and laxity distributions.

For each combination the simulation ran until it reached a confidence level 95% in the results for a maximum error (e.g., one half of the confidence interval) of (1) 2% of the specified probability if P_{dyn} is the measure of interest, (2) 0.2% of the specified response time value if response time is the measure, (3) 5% of the task arrival rate if the maximum system utilization is the measure, and (4) 5% of the ratio, frequency, or fraction value if task transfer–out ratio, frequency of broadcasts/state probes, or fraction of idle time, is the

[8] A system is said to be *regular* if all node degrees are identical.

[9] For convenience, $E(R)$ is normalized to 1.

measure.[10]

We first determine the tunable parameters used in the proposed scheme. Second, we evaluate and compare different schemes with respect to several important performance metrics obtained from the simulations. Then, we analyze the effect of (1) varying processing/communication overheads, (2) using FCFS (instead of MLFS) as the local scheduling discipline, (3) using QL (instead of CET) as the measure of workload on these LS schemes, (4) excluding the Bayesian decision analysis from the proposed scheme, and (5) statistical fluctuation in task arrival patterns (represented by bursty task arrivals) on the performance of the proposed LS scheme.

3.5.2 Determination of Tunable Parameters in the Proposed Scheme

The accuracy of prior/posterior distributions depends on the values of such tunable parameters as the probability update interval T_p, the probability update ratio a, and the number (K_r) and values of state–region thresholds. It is, however, difficult to objectively determine an optimal combination of these parameters which will give accurate prior/posterior distributions while incurring the least overhead. The main reasons for this are:

- The choice of a and T_p depends on the application–dependent variation of workload.
- The number and values of thresholds must be determined by optimizing the tradeoff between the resolution of state–region division and the overhead of the resulting region–change broadcasts. It is impossible to determine the optimal number and values of state–region thresholds without a closed–form expression for this tradeoff. Moreover, the optimal number and values of thresholds depend on both the laxity and execution time distributions of the task set.

Thus, we shall determine the tunable parameters for each task set with the following two steps:

S1. We first fix all but one parameter of interest at a time, and obtain the performance curve as a function of this parameter from which its optimal value can be determined. Next, we vary another parameter of interest while keeping the first parameter fixed at its optimal value and the rest of the parameters fixed at their originally–chosen values. This process will be repeated until all the parameters have been varied.

[10]The number of simulation experiments needed to achieve the above confidence interval is calculated based on the assumption that the parameter to be estimated/measured has a normal distribution with unknown mean and variance.

S2. Since a different order of examining parameters in **S1** may lead to different results, there may may be more than one parameter set from which we choose the one with the smallest P_{dyn} and at the same time, reasonably small processing/communication overheads (e.g., the processing power used for updating distributions and calculating Bayesian decisions, the frequency of region–change broadcasts, or the task transfer–out ratio) as the 'optimal' parameter set.

The sets of parameters obtained through the above two steps may not be globally optimal, but our simulation results have shown them to yield good results as compared to other schemes. Moreover, as will be shown later, our simulation results indicate that the proposed scheme is robust to the variation of the tunable parameters. Thus, the result with a set of sub-optimal parameters can still give a representative performance of the proposed scheme.

Summarized below are some of the important findings in determining the tunable parameters. The task set with the task arrival rate $\lambda^{ext} = 0.8$, the distribution of task execution time $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and the distribution of task laxity $L = \{1.0, 2.0, 3.0\}_{1/3}$ is arbitrarily chosen for an illustrative purpose.

Probability Update Interval T_p and Probability Update Ratio a: Frequent probability updates generate more accurate distributions and thus provide each node with better knowledge of the inconsistency between its observation of other nodes and the corresponding true states. This benefit must be weighed against the associated overheads. As shown in Table 3.1, P_{dyn} for the proposed scheme with the threshold pattern {0.2, 0.6, 1.0} decreases first as T_p increases from 10 to 100 (in units of the average task execution time), and then increases as T_p increases further beyond 100. (The task set with $\lambda^{ext} = 0.4$ exhibits a similar behavior except that T_p which results in the smallest P_{dyn} is slightly increased.) The probability update ratio a is chosen to be 0.5, since the task arrival rate is constant over time, and thus, the distribution constructed from the state samples gathered before is as good as that constructed from the state samples collected over the last T_p units of time.

Selection of the default probability distributions (or equivalently, the default risk function ζ^{T_d} in Eq. (3.6)) at the system startup is shown to have only minor effects on system performance as long as T_p is properly chosen. So, the initial posterior distribution of CET is chosen to be uniform. The default risk function can then be expressed as:

$$\zeta^{T_d}(P_{W|O=\underline{o}}, d_i) \;=\; P_{W_i|O_i}(W_i > T_d \mid O_i = o_i)$$

$$= \begin{cases} 0, & T_d \geq TH_{o_i+1}, \\ 1, & T_d < TH_{o_i}, \\ \frac{TH_{o_i+1}-T_d}{TH_{o_i+1}-TH_{o_i}}, & \text{otherwise}, \end{cases}$$

where $i \geq 0$ and $TH_0 \triangleq 0$.

Threshold Patterns: We first analyze the effect of changing the resolution of state–space division with the number of regions fixed at 4. As shown in Table 3.2 (a), P_{dyn} decreases as the interval between two thresholds gets smaller, which will henceforth be called the *threshold interval*. (For convenience, all threshold intervals are assumed to be identical; even if this assumption does not hold, the following discussion will remain the same except for more complicated descriptions.) However, the decrease in P_{dyn} becomes insignificant when the threshold interval gets smaller than the least task execution time within the task set. Thus, the threshold interval must not be smaller than the least task execution time. Once the threshold interval is selected, one can analyze the effects of changing TH_1 (and thus, other thresholds). It is shown in Table 3.2 (b) that P_{dyn} decreases as TH_1 decreases. The change in P_{dyn} again becomes insignificant as TH_1 gets smaller than the least task execution time. Note that in our proposed scheme, selecting $TH_1 <$ the least task execution time $< TH_2$ is essentially equivalent to implementing the shortest queue policy in [ELZ86] except that the ways of collecting state information are different (the latter with state probing, and the former with region–change broadcasting). Thus, a modified shortest queue policy (or $TH_1 \leq$ the least task execution time) is preferred for the proposed scheme to minimize P_{dyn}.

One interesting phenomenon is that the frequency of region–change broadcasts is relatively high when the thresholds coincide with the task execution times. This is because the acceptance/completion of a task makes a node's CET easily cross a certain threshold, thus resulting in an excessive number of region–change broadcasts. Such type of thresholds are ruled out. Thus, to implement the shortest queue policy while avoiding excessive broadcasts, TH_1 must be slightly smaller than the least task execution time.

Number of State Regions: To analyze the effects of the number of state regions on system performance, we ran simulations while changing the number of regions from 2 to 6. A node broadcasts the change of state region (i) whenever the node's CET crosses TH_1 if only two state regions are used, and (ii) whenever the node's CET crosses TH_{2k} if $K_i \geq 3$ state regions are used, where $1 \leq k \leq \lceil \frac{K_i}{2} \rceil - 1$.

T_p	Prob. of dynamic failure	Percentage of idle time
10	8.0564×10^{-3}	0.1874
50	7.9761×10^{-3}	0.1902
100	7.1967×10^{-3}	0.1913
200	7.4627×10^{-3}	0.1946
500	7.8438×10^{-3}	0.1951
1000	7.9582×10^{-3}	0.1964

Table 3.1: T_p for a task set with $\lambda^{ext} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and $L = \{1, 2, 3\}_{1/3}$.

Threshold pattern	P_{dyn}	Task transfer-out ratio, r_{tt}
0.4,0.5,0.6	7.6857×10^{-3}	0.228
0.4,0.6,0.8	7.8406×10^{-3}	0.220
0.4,0.8,1.2	8.0502×10^{-3}	0.214
0.4,1.0,1.6	1.2833×10^{-2}	0.204
0.4,1.2,2.0	1.5104×10^{-2}	0.207

(a): Effects of the threshold interval for a task set with $\lambda^{ext} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and $L = \{1, 2, 3\}_{1/3}$.

Threshold pattern	P_{dyn}	r_{tt}	Freq. of broadcasts f_b
0.1,0.5,0.9	7.1873×10^{-3}	0.230	0.4987
0.2,0.6,1.0	7.1967×10^{-3}	0.216	0.4943
0.4,0.8,1.2	8.0502×10^{-3}	0.214	0.5723
0.6,1.0,1.4	1.0575×10^{-2}	0.213	0.5122
0.8,1.2,1.6	1.2479×10^{-2}	0.211	0.5249

(b): Effects of of TH_1 for a task set with $\lambda^{ext} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and $L = \{1, 2, 3\}_{1/3}$.

Table 3.2: Effects of the number and values of thresholds on P_{dyn}.

No. of state regions	P_{dyn}	Freq. of broadcasts
2	8.0932×10^{-3}	0.5079
3	7.3108×10^{-3}	0.5086
4	7.1967×10^{-3}	0.4943
5	7.1383×10^{-3}	0.5942
6	7.1156×10^{-3}	0.5967

Table 3.3: Effect of the number of state regions on P_{dyn}. Task set: $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$.

As shown in Table 3.3, P_{dyn} decreases as the number of state regions grows, and the decrease in P_{dyn} becomes insignificant when the number of regions grows beyond 4. Moreover, the frequency of region-change broadcasts remains essentially unchanged when the number of regions is below 4, but increases as the number of regions grows beyond 5. This is because a finer resolution of state intervals results in more reference points for region-change broadcasts. Thus, 3 or 4 state regions suffice to give a satisfactory performance.

One significant result is that the proposed scheme is shown to be robust to the variation of the tunable parameters, as compared to the other schemes reported in [SC89a, MTS89a, PTS88]. The change in P_{dyn} is shown to be less than 10^{-3} for any given change in either the threshold interval, or the number of state regions, or the values of thresholds. This is an important advantage coming from the use of prior/posterior distributions and Bayesian analysis. The proposed LS scheme can thus provide good performance even with not well-tuned parameters as long as the general rules discussed above are followed.

3.5.3 Performance Evaluation with respect to Different Measures

Probability of Dynamic Failure

A task is said to be *missed* and dynamic failure occurs if the sum of its queueing-for-execution time and the delay in transferring the task exceeds its laxity. Let $P_{dyn|\ell}$ denote the probability of missing deadlines for a task with laxity ℓ. Then,

$$P_{dyn} = \sum_{\ell=0}^{L_{max}} P_{dyn|\ell} \cdot \hat{q}_\ell.$$

Figs. 3.2 and 3.3 are the plots of P_{dyn} vs. task arrival rate (λ^{ext}), and $P_{dyn|\ell}$ vs. task laxity ℓ, respectively. Table 3.4 shows some numerical results of $P_{dyn|\ell}$ under different schemes. As was expected, P_{dyn} increases as the system load gets heavy and/or the task laxity gets tight.

The random selection scheme outperforms the state probing scheme when the system load gets heavy (e.g., Table 3.4 (a) vs. (b)) or the task laxity gets tight (e.g., $L = \{1, 2, 3\}$ vs. $L = \{1\}$ in Table 3.4 (b)). This is because (1) under heavy loads, most nodes are likely to become unable of completing tasks in time, which will in turn make state probing unsuccessful most of the time, and (2) probing other nodes before sending an overflow task requires two communication messages (one for request and the other for response), whereas the random selection does not require such messages. This negative effect becomes more pronounced as laxities get tighter.

The focused addressing scheme outperformed the state probing and random selection schemes, but was inferior to the proposed scheme, especially when the task laxity is tight. This is because:

- The focused node or its successor node — the node that the focused node will retransfer the task to — among those 'seen' capable is basically chosen randomly, thus increasing the chance of two nodes sending their overflow tasks to the same node.
- Not many RFB messages are issued under light loads, thus making a node unable to keep its observation of other nodes up-to-date and increasing the chance of transferring a task to an incapable focused node. This is intolerable to tasks with tight laxities.
- Requests and replies for bids become excessive under heavy loads, thus increasing communication delays. The state information collected via periodic state exchanges or the bids sent from other nodes may thus become out-of-date.

In all cases simulated, the proposed LS scheme is shown to outperform all but the perfect information scheme in meeting task deadlines, showing its effectiveness achieved by judicious collection and use of state information. It does not perform as well as the perfect information scheme due to the additional processing overhead introduced by the probability update process and the communication delays incurred in task transfers and region-change broadcasts. See Table 3.5 for the case where all processing/communication overheads are set to zero while holding the task-transfer delay at 10% of task execution time. If the processing/communication overheads were ignored, all the realistic schemes would perform much better than they actually do, thus under-estimating P_{dyn} — which is undesirable for real-time applications.

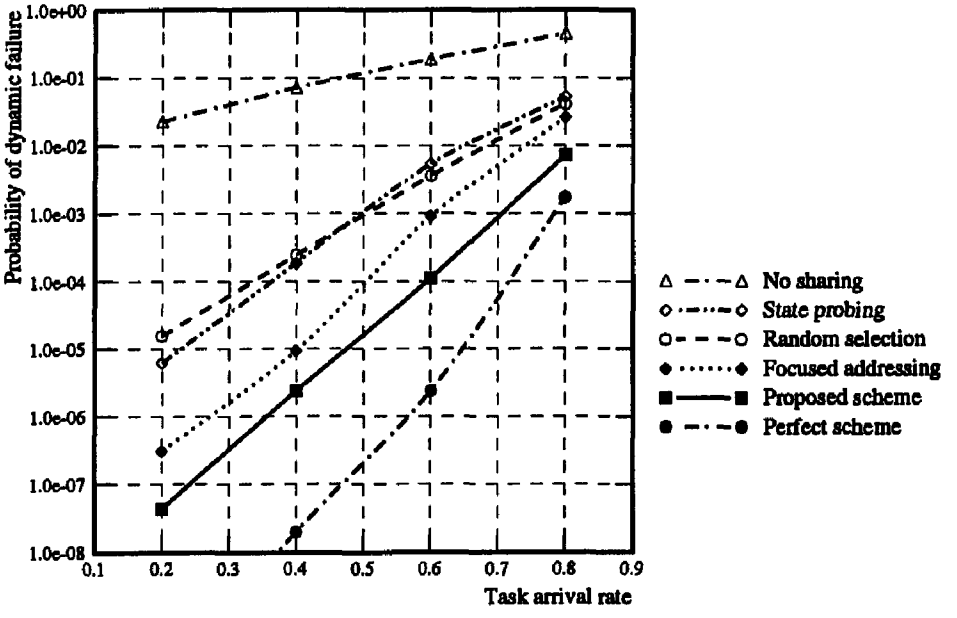


Figure 3.2: P_{dyn} vs. task arrival rate for a 16–node system with a task set: $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{1/3}$.

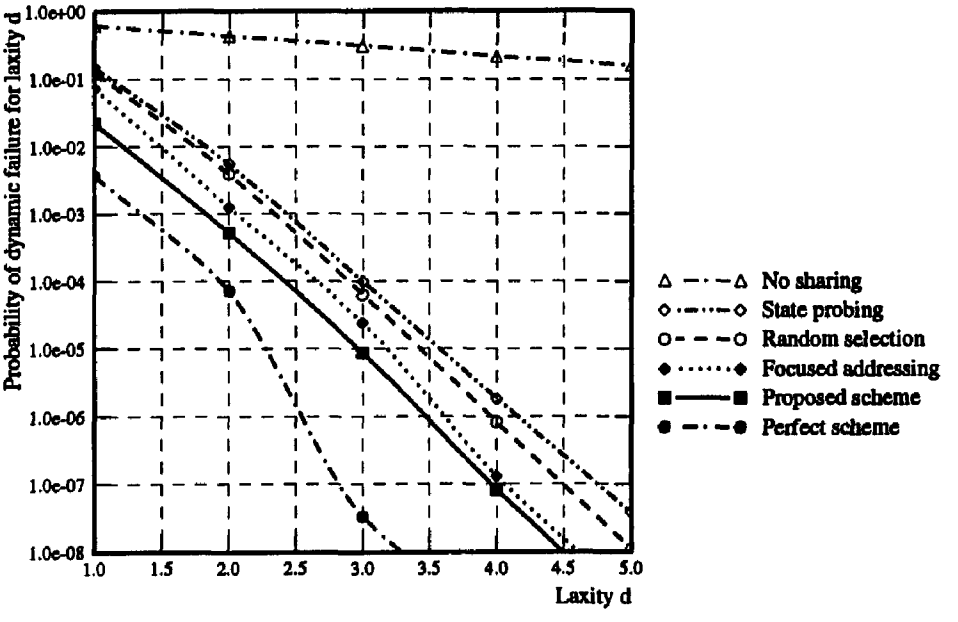


Figure 3.3: $P_{dyn|\ell}$ vs. task laxity ℓ for a 16–node system with a task set: $\lambda^{ext} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3, 4, 5\}_{0.2}$).

($\lambda^{ext} = 0.8$) Task attributes	Lax. ℓ	No sharing	State probing	Random selection	Focused addressing	Proposed scheme	Perfect scheme
$ET = \{0.4, 0.8,$	1	0.6107	0.1515	0.1214	8.649×10^{-2}	2.123×10^{-2}	5.093×10^{-3}
$1.2, 1.6\}_{0.25},$	2	0.4317	4.779×10^{-3}	2.162×10^{-3}	9.746×10^{-4}	3.523×10^{-4}	6.492×10^{-5}
$L = \{1, 2, 3\}_{1/3}$	3	0.3058	3.514×10^{-5}	1.231×10^{-5}	1.026×10^{-5}	7.828×10^{-6}	5.721×10^{-8}
$ET = \{0.027,$	1	0.7059	0.2476	0.1992	0.1524	4.043×10^{-2}	3.090×10^{-2}
$0.27, 2.703\}_{1/3},$	2	0.6169	5.086×10^{-2}	3.372×10^{-2}	2.249×10^{-2}	7.819×10^{-3}	2.539×10^{-3}
$L = \{1, 2, 3\}_{1/3}$	3	0.5061	4.994×10^{-3}	2.860×10^{-3}	9.594×10^{-4}	4.793×10^{-4}	1.763×10^{-4}
$ET = \{0.4, 0.8,$ $1.2, 1.6\}_{0.25},$ $L = \{1\}$	1	0.6075	0.1293	8.016×10^{-2}	7.153×10^{-2}	2.583×10^{-2}	6.012×10^{-3}

(a) $\lambda^{ext} = 0.8$.

($\lambda^{ext} = 0.4$) Task attributes	Lax. ℓ	No sharing	State probing	Random selection	Focused addressing	Proposed scheme	Perfect scheme
$ET = \{0.4, 0.8,$	1	0.1612	3.594×10^{-4}	7.293×10^{-4}	8.264×10^{-5}	1.391×10^{-5}	5.892×10^{-8}
$1.2, 1.6\}_{0.25},$	2	0.0421	5.402×10^{-6}	1.262×10^{-5}	9.536×10^{-7}	2.930×10^{-7}	1.583×10^{-10}
$L = \{1, 2, 3\}_{1/3}$	3	0.0117	1.782×10^{-7}	4.296×10^{-7}	4.846×10^{-8}	7.497×10^{-9}	0
$ET = \{0.027,$	1	0.2854	2.270×10^{-3}	5.669×10^{-3}	9.079×10^{-4}	2.105×10^{-4}	3.516×10^{-6}
$0.27, 2.703\}_{1/3},$	2	0.1761	2.346×10^{-5}	1.053×10^{-4}	9.896×10^{-6}	2.970×10^{-6}	2.835×10^{-8}
$L = \{1, 2, 3\}_{1/3}$	3	0.0815	2.693×10^{-7}	1.775×10^{-6}	7.903×10^{-8}	1.863×10^{-8}	0
$ET = \{0.4, 0.8,$ $1.2, 1.6\}_{0.25},$ $L = \{1\}$	1	0.1660	8.163×10^{-4}	6.250×10^{-4}	3.818×10^{-4}	7.018×10^{-5}	9.476×10^{-8}

(b) $\lambda^{ext} = 0.4$.

Table 3.4: $P_{dyn|\ell}$ vs. task laxity ℓ for different task sets under different schemes ($N = 16$).

Arrival rate (λ^{ext})	Laxity ℓ	No sharing	State probing	Random selection	Focused addressing	Proposed scheme	Perfect scheme
0.8	1	0.6107	3.027×10^{-2}	4.841×10^{-2}	2.878×10^{-2}	2.121×10^{-2}	5.093×10^{-3}
	2	0.4317	2.937×10^{-4}	3.161×10^{-4}	2.874×10^{-4}	2.688×10^{-4}	6.492×10^{-5}
	3	0.3058	8.763×10^{-7}	8.754×10^{-6}	5.323×10^{-7}	1.168×10^{-7}	5.721×10^{-8}
0.4	1	0.1612	1.875×10^{-7}	9.372×10^{-5}	4.275×10^{-7}	2.034×10^{-7}	5.892×10^{-8}
	2	0.0421	8.764×10^{-8}	2.167×10^{-6}	9.619×10^{-8}	1.457×10^{-8}	1.583×10^{-10}
	3	0.0117	4.763×10^{-10}	1.928×10^{-8}	5.136×10^{-10}	0	0

Table 3.5: $P_{dyn|d}$ for a task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$ under the ideal condition.

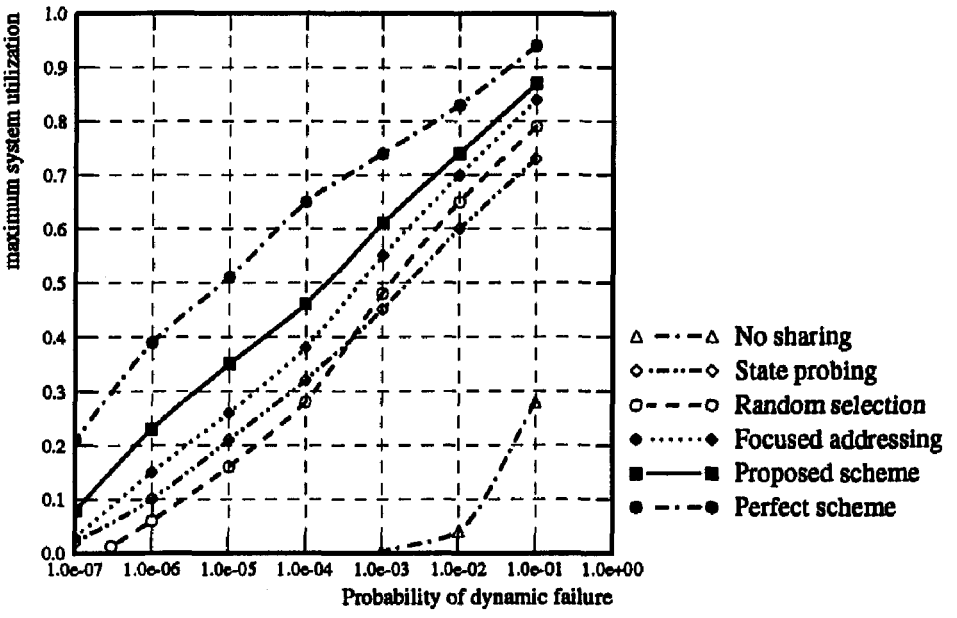


Figure 3.4: λ_{max} vs. P_{dyn} for a 16–node system.

Maximum System Utilization

The system utilization is defined as the ratio of the external task arrival rate (λ^{ext}) to the system service rate ($1/E(R)$). The service rate is normalized to 1 in our analysis, and thus, the system utilization simply becomes λ^{ext}. Since P_{dyn} increases with system load (Fig. 3.2), there exists an upper bound for λ^{ext}, termed as *maximum system utilization* λ_{max}, below which $P_{dyn} \leq \epsilon$ can be guaranteed for some pre-specified $\epsilon > 0$. Fig. 3.4 shows plots of the maximum system utilization vs. ϵ for a 16–node regular system. One important result is that we do not have to sacrifice system utilization to lower P_{dyn}, which is in contrast to the common notion of trading system utilization for real–time performance. Moreover, the proposed scheme has the performance closest, among all LS schemes considered, to the perfect information scheme, and usually outperforms the other realistic schemes by almost an order of magnitude.

Mean Response Time

Conventionally, mean response time (MRT) is used as a global system performance index in non–real-time distributed systems, and there have been many approaches to the goal of minimizing MRT. Table 3.6 gives MRT for the various task attributes under different schemes. MRT increases as the system load increases, or the variance of either the

distribution of task execution time or the distribution of task laxity gets large.

One interesting result is that the MRT associated with the proposed scheme varies least drastically with the change of distributions of task laxity and/or task execution time as compared to those associated with other LS schemes. This is due to the use of Bayesian analysis to choose a receiver node for task transfer. Moreover, our LS scheme outperforms all but the perfect information scheme even when all the processing overheads are taken into account and MRT is used as the performance index. This is due to the fact that, to minimize P_{dyn}, the proposed scheme aims to share overflow tasks among all capable nodes in the system, rather than only within a buddy set (by using the preferred lists and overlapping buddy sets [SC89a]), thus spreading loads throughout the entire system. On the other hand, a scheme that results in a lower MRT does not always yield a lower P_{dyn}. For example, consider the case when $\lambda^{ext} = 0.4$ in Table 3.6 (b) and Table 3.4 (b). The state probing scheme has a larger average task MRT than the random selection scheme, but a smaller P_{dyn}. Consequently, those approaches based on minimizing MRT *alone* may not be directly applicable to real-time systems.

Task Transfer–Out Ratio

A task arrived at a node has to be transferred if and only if the CET of that node exceeds the laxity of the task. The task transfer ratio, r_{tt}, is defined as the portion of arrived tasks (both external and transferred–in tasks) that must be transferred. r_{tt} is a measure of the traffic overhead caused by task transfers. Table 3.7 shows the values of r_{tt} for the various task attributes under different schemes.

Under light to medium loads (e.g., $\lambda^{ext} = 0.2$—0.4 in Table 3.7), the task transfer ratios associated with different schemes are very close to one another. This is because most tasks can be completed in time locally or with at most one task transfer, and thus, the location policies employed by different schemes have little influence on system performance. On the other hand, when the system load gets heavy, the way of choosing a node for task transfer/re-transfer results in notable differences in performance. The state probing scheme has the task transfer ratio closest to the perfect information scheme, since it first checks a node's LS capability before transferring a task to that node. The proposed scheme is inferior to the state probing scheme due to its use of imperfect observation to probabilistically decide the receiver node. However, it does not require any probing overhead at the time of making a location decision.

The focused addressing scheme performs slightly better than the proposed scheme under light load. However, when the system load gets heavy, the performance of the focused

($\lambda^{ext} = 0.8$) Task attributes		No sharing	State probing	Random selection	Focused addressing	Proposed scheme	Perfect scheme
$L = \{1,2,3\}_{1/3}$	$ET = \{1\}$	2.983	1.813	1.788	1.763	1.725	1.718
	$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	3.420	1.851	1.814	1.789	1.706	1.692
	$ET = \{0.027, 0.27, 2.703\}_{1/3}$	6.003	2.149	2.047	1.979	1.799	1.797
$ET = \{0.4, 0.8,$ $1.2, 1.6\}_{0.25}$	$L = \{1\}$	3.420	1.584	1.565	1.502	1.470	1.429
	$L = \{1,2\}_{0.5}$	3.420	1.768	1.697	1.679	1.627	1.592
	$L = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	3.420	1.800	1.658	1.579	1.440	1.405

(a) MRT for different task sets under different schemes ($\lambda^{ext} = 0.8$).

($\lambda^{ext} = 0.4$) Task attributes		No sharing	State probing	Random selection	Focused addressing	Proposed scheme	Perfect scheme
$L = \{1,2,3\}_{1/3}$	$ET = \{1\}$	1.336	1.262	1.261	1.259	1.257	1.256
	$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	1.397	1.286	1.267	1.265	1.263	1.259
	$ET = \{0.027, 0.27, 2.703\}_{1/3}$	1.825	1.344	1.322	1.314	1.300	1.295
$ET = \{0.4, 0.8,$ $1.2, 1.6\}_{0.25}$	$L = \{1\}$	1.397	1.186	1.173	1.165	1.159	1.158
	$L = \{1,2\}_{0.5}$	1.397	1.245	1.233	1.233	1.230	1.228
	$L = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	1.397	1.202	1.178	1.162	1.159	1.155

(b) MRT for different task sets under different schemes ($\lambda^{ext} = 0.4$).

Table 3.6: Comparison of mean response time among different LS schemes.

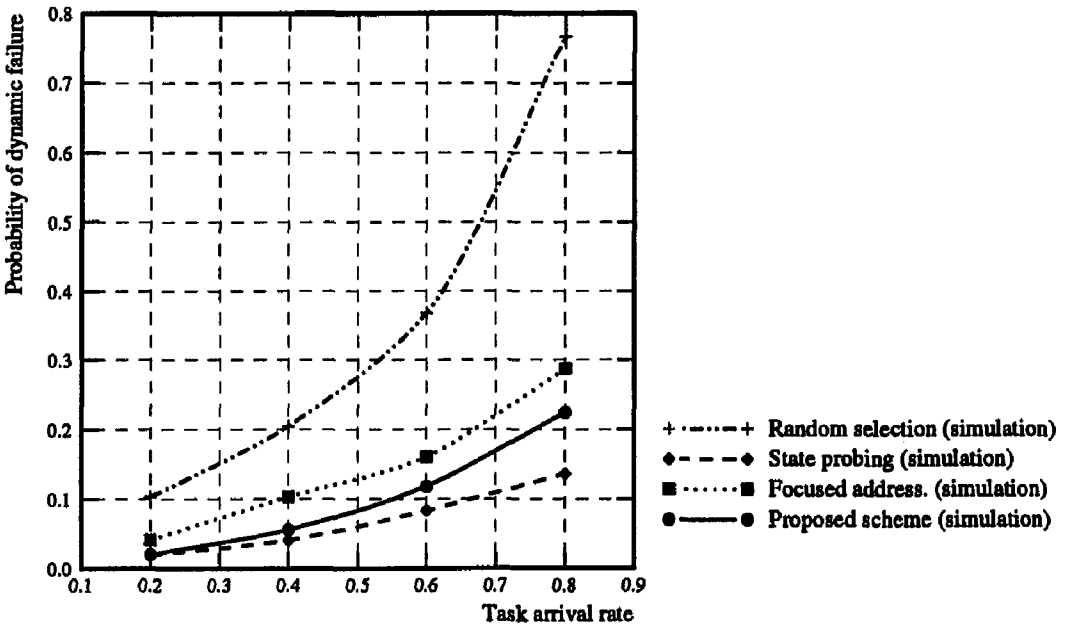


Figure 3.5: Frequency of task collision vs. external task arrival rate for a 16–node system with a task set: $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$.

addressing scheme deteriorates quickly due to the increased probability of making incorrect LS decisions based on out–of–date[11] state information.

Frequency of Task Collision

The frequency of task collision, f_{tc}, is defined as the fraction of transferred tasks that are not guaranteed on remote nodes after their transfer. This is a measure for the capability of the LS algorithms in reducing the probability of task re–transfers. Fig. 3.5 shows the simulation results for different LS schemes.

Generally, f_{tc} increases as the system load gets heavier, the task laxity gets tighter, and/or as the variance of task execution time increases for the same reason that leads to the increase of task transfer–out ratio r_{tt}. The state probing scheme has the lowest frequency of task collision, because it always checks the capability of a node (and thus maintains/uses the most up–to–date state information) before transferring a task. The random selection scheme, on the other hand, does not use any state information for LS decision, and thus necessarily has the highest frequency of task collision. f_{tc} of the proposed scheme lies between that of state probing and that of focused addressing. The reason for the focused

[11] as a result of the increased communication delays caused by excessive bidding messages.

Arrival rate (λ^{ext})	State probing	Random selection	Focused address.	Proposed scheme	Perfect scheme
0.2	0.019	0.021	0.020	0.020	0.019
0.4	0.056	0.065	0.058	0.057	0.053
0.6	0.112	0.152	0.116	0.114	0.107
0.8	0.185	0.333	0.241	0.223	0.184

(a) Task transfer ratio versus task arrival rate for the task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$ under different schemes.

($\lambda^{ext} = 0.8$) Task attributes		State probing	Random selection	Focused address.	Proposed scheme	Perfect scheme
$L = \{1, 2, 3\}_{1/3}$	$ET = \{1\}$	0.160	0.296	0.230	0.224	0.160
	$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	0.185	0.333	0.241	0.223	0.184
	$ET = \{0.027, 0.27, 2.703\}_{1/3}$	0.277	0.526	0.398	0.367	0.276
$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	$L = \{1\}$	0.300	0.463	0.355	0.351	0.286
	$L = \{1, 2\}_{0.5}$	0.223	0.383	0.286	0.268	0.219
	$L = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	0.262	0.449	0.367	0.320	0.280

(b) $\lambda^{ext} = 0.8$.

Table 3.7: Comparison of task transfer ratio among different schemes.

($\lambda^{ext} = 0.4$) Task attributes		State probing	Random selection	Focused address.	Proposed scheme	Perfect scheme
$L = \{1, 2, 3\}_{1/3}$	$ET = \{1\}$	0.036	0.039	0.037	0.038	0.033
	$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	0.056	0.065	0.056	0.057	0.053
	$ET = \{0.027, 0.27, 2.703\}_{1/3}$	0.128	0.160	0.130	0.132	0.123
$ET = \{0.4, 0.8,$ $1.2, 1.6\}$	$L = \{1\}$	0.116	0.130	0.117	0.119	0.111
	$L = \{1, 2\}_{0.5}$	0.076	0.086	0.075	0.076	0.073
	$L = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	0.121	0.152	0.123	0.128	0.120

(c) $\lambda^{ext} = 0.4$.

Table 3.7: (continued) Comparison of task transfer ratio among different schemes.

addressing scheme to be inferior to the proposed scheme is that the state information of other nodes is collected via periodic information exchanges and/or in the bids received in previous RFB activities, and may be obsolete at the time of choosing a focused node. The proposed LS scheme may also use obsolete state information, but it neutralizes the undesirable effect of using out-of-date information with Bayesian decision analysis.

System Size

It is found from the simulations that the larger the system size, the better the performance of the LS schemes will result. (See Table 3.4 and Table 3.8 for numerical results.) This is because a larger system has a larger processing capacity to handle bursty task arrivals at one or more nodes in the system. Besides, task transfers, state probes, RFB,[12] and/or region-change broadcasts do not significantly increase the queueing delay (as compared to a system with only a few nodes) due to the increased communication capacity and the decreased chance of transferring two or more tasks through the same link or to the same node.

Due to the way the buddy sets and the preferred lists constructed [SC89a], the overflow tasks within each buddy set will be evenly shared by all capable nodes in the entire system as the system size grows, rather than overloading a few capable nodes within the same buddy set. Moreover, the preferred list of a node is different from those of other nodes' in its buddy set. Consequently, the percentage of common nodes in the preferred lists of the nodes in a buddy set gets smaller as the system size grows, and thus, there is a greater chance that the overflow tasks of a node may be transferred to some other nodes not in its buddy set, leading to more even distribution of the overflow tasks.

Frequency of Region-Change Broadcasts vs. Frequency of State Probing/Bidding

In the proposed LS scheme, each node has to broadcast a change of state region to all the other nodes in its buddy set. Thus, the frequency of region-change broadcasts, f_b, determines the traffic overhead in collecting state information. On the other hand, the traffic overhead in both the state probing and focused addressing schemes is determined by the frequency of state probing, f_p, and the frequency of RFB, f_r, respectively. Table 3.9 summarizes the simulation results on f_b, f_p, and f_r in terms of number of messages per $E(R)$.

---

[12]To reduce traffic overheads, RFB messages are sent to 10 randomly selected nodes, instead of all the other nodes in the system.

($\lambda^{ext} = 0.8$) Task attributes	Lax. ℓ	No sharing	State probing	Random selection	Focused addressing	Proposed scheme	Perfect scheme
$ET = \{0.4, 0.8,$	1	0.6107	0.1112	8.566×10^{-2}	3.011×10^{-2}	2.326×10^{-3}	2.931×10^{-5}
$1.2, 1.6\}_{0.25},$	2	0.4317	8.541×10^{-4}	3.455×10^{-4}	1.046×10^{-4}	8.798×10^{-6}	8.126×10^{-7}
$L = \{1, 2, 3\}_{1/3}$	3	0.3058	5.472×10^{-6}	6.237×10^{-6}	2.326×10^{-6}	1.266×10^{-7}	4.256×10^{-10}
$ET = \{0.027,$	1	0.7059	0.1687	0.1381	0.1079	1.534×10^{-2}	5.156×10^{-4}
$0.27, 2.703\}_{1/3},$	2	0.6169	1.506×10^{-2}	1.580×10^{-2}	8.956×10^{-3}	2.557×10^{-3}	1.070×10^{-5}
$L = \{1, 2, 3\}_{1/3}$	3	0.5061	2.927×10^{-4}	4.807×10^{-4}	1.046×10^{-4}	2.032×10^{-5}	8.396×10^{-8}
$ET = \{0.4, 0.8,$ $1.2, 1.6\}_{0.25},$ $L = \{1\}$	1	0.6075	9.108×10^{-2}	5.115×10^{-2}	2.674×10^{-2}	5.335×10^{-3}	7.815×10^{-6}

(a) $\lambda^{ext} = 0.8$.

($\lambda^{ext} = 0.4$) Task attributes	Laxity ℓ	No sharing	State probing	Random selection	Focused addressing	Proposed scheme	Perfect scheme
$ET = \{0.4, 0.8,$	1	0.1612	1.203×10^{-4}	6.099×10^{-4}	4.812×10^{-5}	1.169×10^{-5}	6.702×10^{-10}
$1.2, 1.6\}_{0.25},$	2	0.0421	9.320×10^{-7}	2.489×10^{-6}	2.872×10^{-7}	5.148×10^{-8}	0
$L = \{1, 2, 3\}_{1/3}$	3	0.0117	2.301×10^{-8}	5.320×10^{-8}	4.166×10^{-9}	3.049×10^{-10}	0
$ET = \{0.027,$	1	0.2854	9.263×10^{-4}	3.715×10^{-3}	3.146×10^{-4}	1.083×10^{-4}	1.962×10^{-8}
$0.27, 2.703\}_{1/3},$	2	0.1761	5.856×10^{-6}	2.347×10^{-5}	2.021×10^{-6}	1.837×10^{-7}	7.136×10^{-10}
$L = \{1, 2, 3\}_{1/3}$	3	0.0815	2.153×10^{-8}	9.873×10^{-8}	1.041×10^{-8}	5.352×10^{-9}	0
$ET = \{0.4, 0.8,$ $1.2, 1.6\}_{0.25},$ $L = \{1\}$	1	0.1660	5.468×10^{-4}	2.092×10^{-4}	1.012×10^{-4}	5.848×10^{-6}	0

(b) $\lambda^{ext} = 0.4$.

Table 3.8: $P_{dyn|\ell}$ versus task laxity ℓ for different task sets under different schemes ($N = 64$).

Task arrival Rate (λ^{ext})	State probing (f_s)	Focused addressing (f_r)	Proposed scheme (f_b)
0.2	0.0041	0.0178	0.2948
0.4	0.0287	0.2458	0.4517
0.6	0.1144	0.6821	0.4810
0.8	0.4836	1.2346	0.4943

(a) Frequency of state information collection vs. different λ^{ext} for a task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$.

($\lambda^{ext} = 0.8$) Task attributes	State probing (f_p)	Focused addressing (f_r)	Proposed scheme (f_b)
$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{1/3}$	0.4836	1.2346	0.4943
$ET = \{0.027, 0.27, 2.703\}_{1/3}$, $L = \{1, 2, 3\}_{1/3}$	1.0150	1.3816	0.4781
$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1\}$	0.9061	1.5023	0.6872

(b) Frequency of state information collection versus different task sets when $\lambda^{ext} = 0.8$.

Table 3.9: Comparison of the traffic overhead associated with collecting state information between the state probing and the proposed schemes.

Under light to medium loads (e.g., $\lambda^{ext} = 0.2$—0.6 in Table 3.9 (a)), the proposed scheme introduces more traffic overhead (in the worst case, about 0.5 broadcast per $E(R)$) than the other two schemes. However, the additional traffic introduced by broadcasts may not impede the transmission of tasks and/or other messages, since only about 2%—13% of the arrived tasks (Table 3.7) are transferred to other nodes. Besides, the effect of the increased communication delay (as a result of broadcasts) on the inconsistency between a node's observed and true states of other nodes is taken care of by Bayesian analysis.[13] When the system load is heavy, the state probing scheme and the focused addressing scheme perform worse than the proposed scheme (Table 3.9 (b)). This phenomenon becomes more pronounced when the variance of task execution time is large or when the task laxity is tight.

[13]This is evidenced by the fact that the queueing delays due to state broadcasts/task transfers as well as the processing overheads required for probability update/state broadcasts were all included in our simulation.

Processing Overheads due to Region–Change Broadcasts and Probability Updates

Each node updates, once every T_p units of time, the posterior CET distributions given the observation of other nodes using the state samples gathered via region–change broadcasts. The overhead for processing broadcast messages (updating probability distributions) is determined by the broadcast processing cost C_b (the probability–update processing cost C_p), and the frequency of region–change broadcasts f_b (the frequency of probability updates, $1/T_p$). To study the effect of varying the processing overhead, we ran simulations with C_b = 0.5%, 1.0%, 2.5% of average task execution time, C_p = 1%, 2%, 10% of average task execution time, and $1/T_p$ = 0.02, 0.01, 0.001 per $E(R)$, respectively, and computed the portion of simulation time during which a node is found to be idle. As shown in Table 3.10, both C_p and $1/T_p$ have only minor effects on the portion of idle time, while C_b has a slightly larger effect on the portion of idle time. This is expected since region–change broadcasts occur more often (Table 3.10) than probability updates. Under all circumstances, no more than 2% of the processing power is used for region–change broadcasts and probability distribution updates. Thus, these LS mechanisms do not deprive other non-critical tasks of processing power.

FCFS vs. MLFS

Under the MLFS local scheduling discipline, the task queue at each node is ordered by task laxities, and a task with the minimum laxity on the node is always executed first. Another commonly–used local scheduling discipline is the first–come–first–served (FCFS) discipline. A newly–arrived task is added at the end of task queue if it can be completed in time on that node, and will otherwise be considered for transfer. The FCFS discipline is simple and ensures not to alter the existing guarantees, and the transfer policy under this discipline becomes a simple dynamic threshold type [WM85, ELZ86, MTS89a, AC88].[14]

Although the MLFS discipline does not always perform better than the other on a per–task basis, it is shown in [HTT89] to perform better *on the average* in reducing P_{dyn}. To quantitatively compare the performance of these two disciplines, we ran simulations with MLFS and FCFS as the local scheduling discipline for a wide range of task attributes. As shown in Table 3.11, all schemes with FCFS perform worse than their counterparts with MLFS. No matter which local scheduling discipline is used, the proposed scheme is

[14]The threshold may change dynamically with the current state of the node and the time constraints of tasks.

Arrival rate λ^{ext}	Broadcasting cost C_b	Proposed scheme	State probing	Perfect scheme	
				simulation	analytic
0.8	0.5%	0.1930	0.1967	0.1978	0.20
	1.0%	0.1903	0.1944		
	2.5%	0.1839	0.1851		
0.4	0.5%	0.5974	0.5988	0.6014	0.60
	1.0%	0.5950	0.5967		
	2.5%	0.5852	0.5943		

(a) Effect of broadcast cost on the proportion of simulation idle time.

Arrival rate λ^{ext}	Prob. updating cost C_p	Proposed scheme	State probing	Perfect scheme	
				simulation	analytic
0.8	1.0%	0.1924	0.1944	0.1978	0.20
	2.0%	0.1903			
	10.0%	0.1892			
0.4	1.0%	0.5953	0.5967	0.6014	0.60
	2.0%	0.5950			
	10.0%	0.5840			

(b) Effect of probability update cost on the proportion of simulation idle time.

Arrival rate λ^{ext}	Prob. updating freq. $1/T_p$	Proposed scheme	State probing	Perfect scheme	
				simulation	analytic
0.8	0.02	0.1902	0.1944	0.1978	0.20
	0.01	0.1903			
	0.001	0.1964			
0.4	0.02	0.5942	0.5967	0.6014	0.60
	0.01	0.5950			
	0.001	0.5860			

(c) Effect of probability update frequency on the proportion of simulation idle time.

Table 3.10: Comparison of computation overheads for a task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$ between the state probing scheme and the proposed scheme.

Task set	Local scheduling discipline	State probing	Random selection	Focused addressing	Proposed scheme
I	MLFS	5.209×10^{-2}	4.119×10^{-2}	2.582×10^{-2}	7.194×10^{-3}
	FCFS	6.402×10^{-2}	5.594×10^{-2}	3.876×10^{-2}	1.747×10^{-2}
II	MLFS	6.017×10^{-6}	1.736×10^{-5}	3.247×10^{-7}	4.124×10^{-8}
	FCFS	6.204×10^{-6}	1.804×10^{-5}	3.329×10^{-7}	4.236×10^{-8}
III	MLFS	6.013×10^{-2}	5.298×10^{-2}	3.464×10^{-2}	1.343×10^{-2}
	FCFS	6.576×10^{-2}	5.874×10^{-2}	4.187×10^{-2}	1.892×10^{-2}

Table 3.11: Performance (P_{dyn}) comparison of using FCFS/MLFS as the measure of workload. Task set I: $\lambda^{ext} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and $L = \{1, 2, 3\}_{1/3}$. Task set II: $\lambda^{ext} = 0.2$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and $L = \{1, 2, 3\}_{1/3}$. Task set III: $\lambda^{ext} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and $L = \{1.4, 1.5, 1.6\}_{1/3}$.

shown to outperform the other realistic schemes in meeting deadlines. This demonstrates the quality of the location policy in the proposed scheme which uses both Bayesian analysis and preferred lists.

One interesting result is that the performance degradation of the FCFS discipline is not so significant when (1) the system load is light, and thus, the task queue is often short (e.g., task set II in Table 3.11), or (2) task laxities are not distributed very widely (e.g., task set III in Table 3.11); that is, either all laxities are very large or all are very tight. This makes the quality of local scheduling less important in meeting task deadlines.

CET vs. QL as the Measure of Workload

For ease of analysis, the number of tasks queued for execution on each node, or QL, is often adopted as the load state of a node [SC89a, NH85, YL84, WM85, ELZ86, HL86, MTS89a]. This may, however, become inappropriate for real-time applications, because it is the cumulative execution time (CET) on a node that determines whether or not a task can be completed in time on that node. A node with only a few tasks queued may not be able to complete a newly-arrived task in time if a large amount of time is required to complete each queued task. On the other hand, a node with a large QL may still be capable of completing an arrived task as long as the total CET on that node does not exceed the laxity of the task. To show the inappropriateness of QL as the measure of workload, we ran simulations with both QL and CET as the state of a node. As shown in Table 3.12, the performance of the proposed scheme with QL is close to (and sometimes worse than[15])

[15]Now, the proposed scheme essentially degenerates to the random selection scheme but with (1) imperfect QL state information which correlates CET in an unpredictable manner and (2) overheads due to the region-

($\lambda^{ext} = 0.8$) Task attributes	Laxity ℓ	Random selection (state:CET)	Proposed scheme (state:QL)	Proposed scheme (state:CET)
$ET = \{0.4, 0.8,$	1	0.1214	1.085×10^{-2}	2.123×10^{-2}
$1.2, 1.6\}_{0.25}$,	2	2.162×10^{-3}	1.625×10^{-3}	3.523×10^{-4}
$L = \{1, 2, 3\}_{1/3}$	3	1.231×10^{-5}	9.604×10^{-6}	7.828×10^{-6}
$ET = \{0.027,$	1	0.1992	0.2286	4.043×10^{-2}
$0.27, 2.703\}_{1/3}$,	2	3.372×10^{-2}	4.924×10^{-2}	7.819×10^{-3}
$L = \{1, 2, 3\}_{1/3}$	3	2.860×10^{-3}	3.142×10^{-3}	4.793×10^{-4}
$ET = \{0.4, 0.8,$ $1.2, 1.6\}_{0.25}$, $L = \{1\}$	1	8.016×10^{-2}	7.606×10^{-2}	2.583×10^{-2}

Table 3.12: Performance comparison of using CET/QL as the measure of workload.

that of the random selection scheme. The effect of using QL as the measure of workload for other schemes is similar to the proposed scheme. The performance degradation gets severer as the variance of task execution times gets large.

Sensitivity to Communication Delays

There are two types of communication delay to consider: one is the state–collection delay incurred from region–change broadcasts/state probes, where the queueing delays play a dominating role, and the other is the delay associated with task transfers, where both the queueing delays and the transmission delays dominate. To study the effect of communication delays, P_{dyn} was computed with (1) the transmission cost associated with each task transfer being 5, 10, 15, and 20 % of the task execution time, and (2) the queueing delay coefficients being halved, doubled, and tripled. (Recall that the queueing delay coefficients are the coefficients in the linear expression used to model the effect of both the average external task arrival rate and the medium traffic on the queueing delay.)

As shown in Fig. 3.6, the state probing scheme, the random selection scheme, and the focused addressing scheme are all more sensitive to the variation of the transmission cost than the proposed scheme. Also, see Table 3.13 (a) for numerical examples. The performance degradation by the state probing scheme occurs because, as the task transmission delay increases, other tasks may arrive at a probed node during the period between the time it was probed and the time an overflow task (of the probing node) arrives at that node. Thus, there is not much correlation between the state when a node was probed and the

change broadcasts and probability updates.

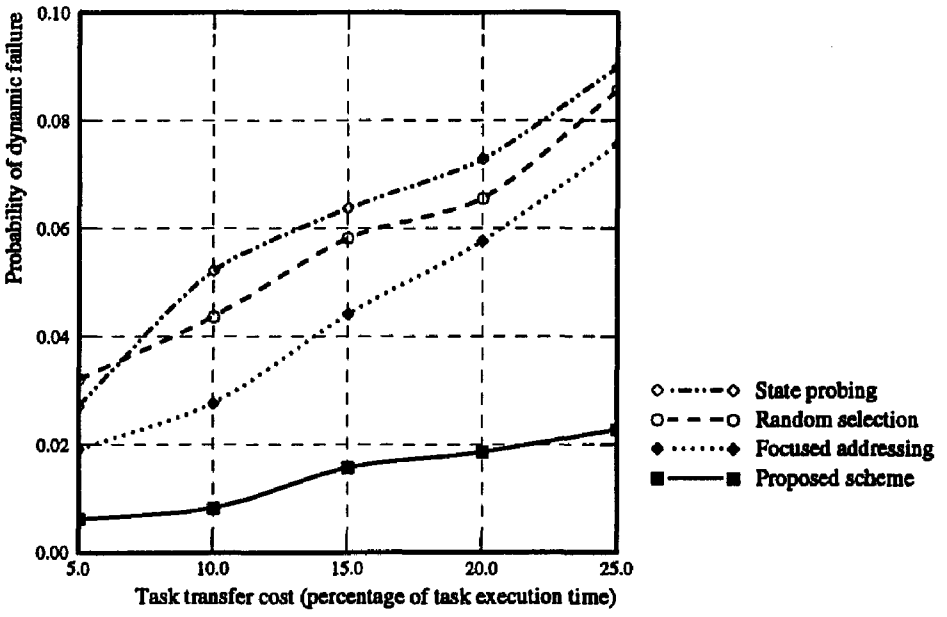


Figure 3.6: P_{dyn} vs. task transfer costs for a 16–node system with a task set: $\lambda^{ext} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{1/3}$).

state when an overflow task arrived at the node. (Similarly, one can reason about the performance degradation of the focused addressing scheme.) The performance of the random selection scheme degrades as the transmission delay increases, due to the combined effect of higher task transfer–out ratios (Table 3.7) and large transmission costs.

Fig. 3.7 (Table 3.13 (b)) shows the effect of varying queueing–related costs on the performance of several LS schemes. The state probing scheme is most sensitive to the variation of queueing delay, because, in addition to suffering the same effect as varying transmission delays, the state probing scheme generates two additional messages per probe, thus increasing the possibility of a task missing its deadline, especially when the queueing delay is large. Varying queueing–related costs has the same effect as varying transmission costs on the random selection scheme as well as on the focused addressing scheme.

In contrast, our proposed scheme is less sensitive to the communication delays (both queueing and transmission delays) because of the use of prior/posterior distributions to characterize the correlation between the observation and the corresponding true state.

Benefit of Using Bayesian Decision Analysis

To actually measure the benefit of using Bayesian decision analysis, we ran a set of experiments using a scheme which is identical in all aspects to the proposed scheme except

($\lambda^{ext} = 0.8$) Trans. costs	Laxity ℓ	State probing	Random selection	Focused addressing	Proposed scheme
5%	1	7.769×10^{-2}	0.1090	5.623×10^{-2}	1.737×10^{-2}
	2	7.614×10^{-4}	3.257×10^{-3}	3.924×10^{-4}	1.989×10^{-4}
	3	3.965×10^{-6}	1.167×10^{-5}	5.311×10^{-6}	3.042×10^{-6}
10%	1	0.1515	0.1214	8.649×10^{-2}	2.123×10^{-2}
	2	4.779×10^{-3}	2.162×10^{-3}	9.746×10^{-4}	3.523×10^{-4}
	3	3.514×10^{-5}	1.231×10^{-5}	1.026×10^{-5}	7.828×10^{-6}
15%	1	0.1834	0.1620	0.1328	3.620×10^{-2}
	2	7.524×10^{-3}	5.261×10^{-3}	2.678×10^{-3}	1.050×10^{-3}
	3	5.851×10^{-5}	2.943×10^{-5}	2.436×10^{-5}	1.746×10^{-5}
20%	1	0.2068	0.1907	0.1708	5.858×10^{-2}
	2	1.154×10^{-2}	7.607×10^{-3}	3.849×10^{-3}	1.604×10^{-3}
	3	1.878×10^{-4}	4.689×10^{-5}	5.014×10^{-5}	2.346×10^{-5}
25%	1	0.2550	0.2408	0.2212	6.128×10^{-2}
	2	1.394×10^{-2}	1.222×10^{-2}	8.746×10^{-3}	3.312×10^{-3}
	3	3.869×10^{-4}	1.054×10^{-4}	9.249×10^{-5}	5.022×10^{-5}

(a) Effect of task transfer costs on $P_{dyn|\ell}$.

($\lambda^{ext} = 0.8$) Queueing coeff.	Laxity ℓ	State probing	Random selection	Focused addressing	Proposed scheme
halved	1	6.091×10^{-2}	8.280×10^{-2}	4.206×10^{-2}	1.831×10^{-2}
	2	3.758×10^{-4}	1.033×10^{-3}	6.270×10^{-4}	3.045×10^{-4}
	3	2.813×10^{-6}	6.948×10^{-6}	5.708×10^{-6}	2.530×10^{-6}
values from simulation	1	0.1515	0.1214	8.649×10^{-2}	2.123×10^{-2}
	2	4.779×10^{-3}	2.162×10^{-3}	9.746×10^{-4}	3.523×10^{-4}
	3	3.514×10^{-5}	1.231×10^{-5}	1.026×10^{-5}	7.828×10^{-6}
doubled	1	0.2134	0.1978	0.1538	3.041×10^{-2}
	2	2.801×10^{-2}	7.552×10^{-3}	1.537×10^{-3}	5.950×10^{-4}
	3	5.513×10^{-4}	1.459×10^{-5}	1.324×10^{-5}	1.167×10^{-5}
tripled	1	0.4194	0.2406	0.2173	4.328×10^{-2}
	2	7.475×10^{-2}	1.450×10^{-2}	1.267×10^{-2}	7.377×10^{-3}
	3	3.842×10^{-3}	1.996×10^{-4}	1.726×10^{-4}	2.348×10^{-5}

(b) Effect of queueing delays on $P_{dyn|\ell}$.

Table 3.13: Effect of communication delays on P_{dyn} for a task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$ under different schemes.

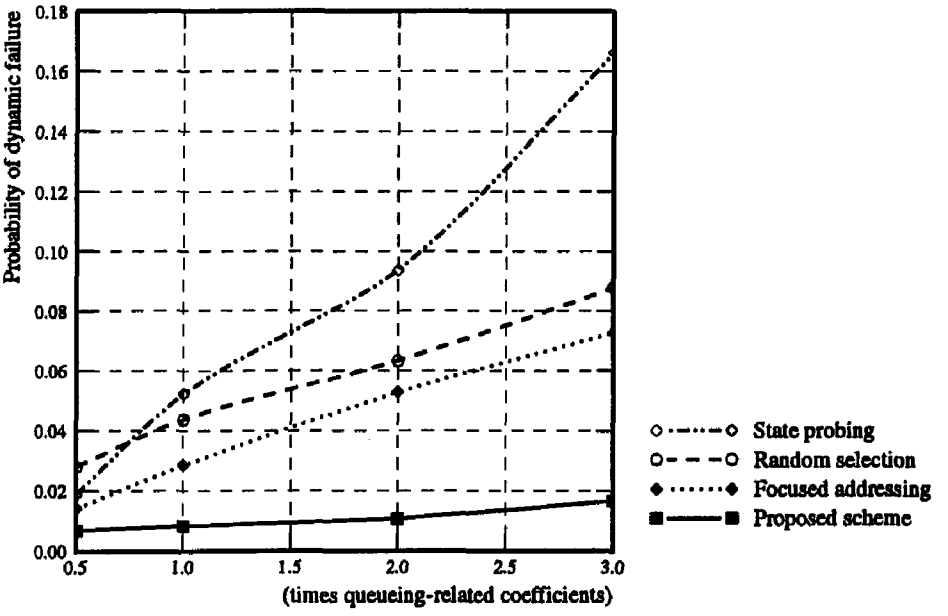


Figure 3.7: P_{dyn} vs. queueing delay coefficients for a 16–node system with a task set: $\lambda^{ext} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{1/3}$).

that no Bayesian analysis is used to capture the inconsistency between the observation and the true state. As the numerical results in Table 3.14 indicate, the proposed scheme with the Bayesian analysis does outperform the one without the Bayesian analysis in minimizing P_{dyn}, especially when

1. task execution times vary over a wide range (Table 3.14 (a)),
2. the distribution of task laxity gets tight (Table 3.14 (a)),
3. the state-collection/task-transfer delays get large (Table 3.14 (b)).

The possibility of 'improper' task transfers as a result of using outdated state information increases under condition 1 and 3, while under condition 2 tasks with tight laxities are less immune to improper task transfers. All these conditions can be handled by characterizing the inconsistency between the true state and the outdated observation with Bayesian analysis.

Statistical Fluctuations in Task Arrivals

One issue in using a Bayesian decision model is to what extent the proposed scheme remains effective when the task arrival pattern randomly fluctuates. This effect is evaluated by simulating different task sets with hyper-exponential interarrival times. This

Task attributes ($\lambda^{ext} = 0.8$)	laxity ℓ	with Bayesian analysis	w/o Bayesian analysis
$ET = \{0.4, 0.8,$	1	2.123×10^{-2}	4.852×10^{-2}
$1.2, 1.6\}_{0.25}$,	2	3.523×10^{-4}	5.272×10^{-4}
$L = \{1, 2, 3\}_{1/3}$	3	7.828×10^{-6}	1.069×10^{-5}
$ET = \{0.027,$	1	4.043×10^{-2}	0.1274
$0.27, 2.703\}_{1/3}$,	2	7.819×10^{-3}	2.134×10^{-2}
$L = \{1, 2, 3\}_{1/3}$	3	4.793×10^{-4}	8.274×10^{-4}
$ET = \{0.4, 0.8,$ $1.2, 1.6\}_{0.25}$, $L = \{1\}$	1	2.583×10^{-2}	6.178×10^{-2}

(a) $P_{dyn|\ell}$ versus task laxity ℓ for different task sets. Task transfer costs are assumed to be 10% of the execution time of the task transferred.

Transfer costs	laxity ℓ	with Bayesian analysis	w/o Bayesian analysis
5%	1	1.737×10^{-2}	2.628×10^{-2}
	2	1.989×10^{-4}	3.046×10^{-4}
	3	3.042×10^{-6}	5.116×10^{-6}
10%	1	2.123×10^{-2}	4.852×10^{-2}
	2	3.523×10^{-4}	5.272×10^{-4}
	3	7.828×10^{-6}	1.069×10^{-5}
20%	1	5.858×10^{-2}	0.1408
	2	1.604×10^{-3}	3.180×10^{-3}
	3	2.346×10^{-5}	4.923×10^{-5}

(b) P_{dyn} vs. task transfer costs for the task set with $\lambda^{ext} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and $L = \{1, 2, 3\}_{1/3}$.

Table 3.14: $P_{dyn|\ell}$ with/without the use of Bayesian analysis in the proposed scheme.

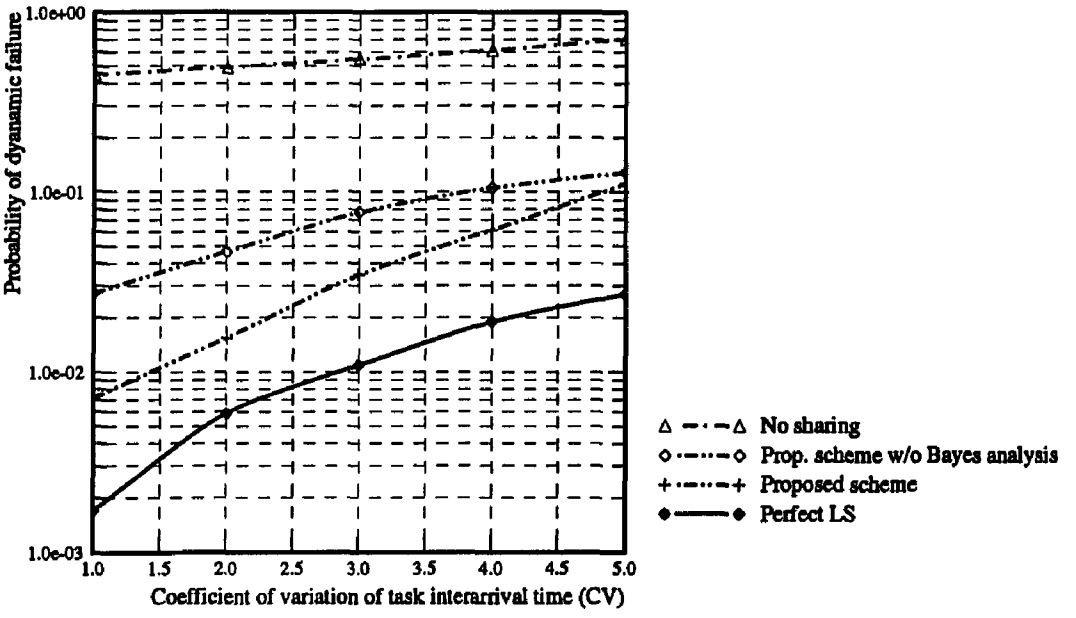


Figure 3.8: P_{dyn} vs. coefficient of variation of task interarrival times for a 16–node system with a task set: $\lambda^{ext} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{1/3}$.

represents a system potentially with bursty task arrivals, and the degree of fluctuation over short periods is modeled well by varying the coefficient of variation (CV) of the hyper-exponential task interarrival times. Specifically, let T_t be the task interarrival time. By Chebyshev's inequality,

$$P(\mid T_t - E(T_t) \mid \geq \ nE(T_t)) \leq \frac{CV^2}{n^2};$$

i.e., the smaller CV^2, the less likely T_t will deviate from its mean, $E(T_t)$. Fig. 3.8 shows the simulation results under heavy system loads ($\lambda^{ext} = 0.8$) where the LS performance is sensitive to the variation of CV. From Fig. 3.8, we draw the following conclusions: (1) the two curves labeled as the proposed scheme and the proposed scheme without using Bayesian analysis give another evidence that LS does benefit from the use of Bayesian decision theory; (2) the performance of the proposed scheme degrades as CV increases. However, the proposed scheme remains effective up to CV=5.42 (or $CV^2 = 30$) beyond which it reduces essentially to the scheme without using Bayesian decision analysis.

3.6 Conclusion

Using prior/posterior distributions and Bayesian analysis, we proposed a new LS scheme which can estimate, even with out-of-date state information, the workload of other

nodes, and select the best candidate receiver of each overflow task. The probability of dynamic failure as a result of using out–of–date information is thus reduced significantly. Moreover, as the simulation results indicate, the ability of making Bayesian decisions based on imperfect state information makes this scheme insensitive to communication delays. The proposed scheme is also shown to be robust to the variation of tunable parameters used in adaptive LS.

In Chapter 4, using the continuous time Markov chain embedded in the corresponding Markov process, we shall develop an analytic model that describes the state evolution of a node with Poisson arrivals for several LS schemes.

CHAPTER 4

ANALYTIC MODELS OF ADAPTIVE LS SCHEMES

4.1 Introduction

In Chapter 3, we proposed, without any modeling analysis, a new decentralized, dynamic LS scheme that uses the CET of each node and combines the preferred lists, region–change broadcasts, and Bayesian analysis both to minimize the probability of dynamic failure and to alleviate the performance degradation caused by communication delays. In this chapter, we develop analytic models for this scheme as well as three other schemes: no LS, LS with random selection, and LS with perfect information. Not only the fundamental differences among the transfer and location policies used by different schemes are addressed, but also the computation/communication overheads in implementing these schemes are included in the analytic models. By taking into account these overheads, the analytic models provide a means of assessing the *absolute* real–time performance of the schemes considered. We derive several performance metrics, such as P_{dyn}, task transfer–out ratio, and maximum system utilization. These metrics are then used to assess the proposed LS scheme against the other schemes.

The first step in developing the analytic LS models is to define the (load) state of a node. For ease of analysis, QL is often used as the node's state [ELZ86, YL84, WM85, HL86, SRC85, SC89a, NH85]. However, as discussed in Chapter 3, performance analysis based on QL would be accurate only if all tasks have an identical, or identically–distributed, execution time, *and* the mean task response time is used as the performance metric. If task execution times are neither identical nor identically–distributed, QL is no longer an adequate measure to characterize the load of a node. For example, a node with only a few tasks queued may not be able to complete a newly arrived task in time if a large amount of time is required to complete each queued task. On the other hand, a node with a large QL may still complete an arrived task in time as long as the total CET of that node does not exceed the laxity of this task. Hence, CET is a better load state in modeling real–time

applications.

Most LS schemes known to date are concerned with minimizing mean response time (MRT) for general-purpose distributed systems, except for those in [Sta85, RSZ89, CL86], where the LS algorithms were evaluated via simulation with respect to either the percentage of tasks lost or the probability of dynamic failure. Little research has been done to analytically evaluate LS schemes for real-time applications: Shin and Chang [SC89a] proposed an embedded Markov chain model, where QL is used as the state of a node. Moreover, the exact solution to this model is very difficult, if not impossible, to obtain. In [KC87], performance models were developed using CET as a node's state, but all tasks are assumed to have an identical deadline. By contrast, we use P_{dyn} as the performance metric, and CET as the load state, and allow both task laxity and task execution time to be drawn from arbitrary probability distributions.

Another point that differentiate our work from others is that most previous work has shown that simple LS algorithms can significantly reduce MRT for general-purpose systems, and the incremental benefits of employing complex LS algorithms become insignificant due to their communication/computation overheads. Using the fraction of tasks lost as the performance metric, Kurose *et al.* [KC87] extended this result to soft real-time systems. However, as both our analytic and simulation results indicate, this extension does not necessarily hold when P_{dyn} is used as the performance metric. Our study show that by making judicious exchange/use of state information, complex schemes—though they incur more computation/communication overheads—achieve notable improvement in reducing P_{dyn} over those simple LS schemes.

The rest of this chapter is organized as follows. Section 4.2 describes the system model used and outlines the operations of the proposed LS scheme. Section 4.3 presents a mathematical model based on continuous-time semi-Markov chains that describe the state evolution of a node under different LS schemes. A two-step iterative algorithm used to solve the queueing model is also given in Section 4.3. The computation and communication overheads incurred in implementing the proposed LS scheme are dealt with in Section 4.4. In Section 4.5, we derive several performance metrics, such as CET distributions, task transfer-out ratios, and P_{dyn} for the schemes under consideration, and comparatively evaluate the proposed LS scheme against others with the derived metrics.[1] This chapter concludes with Section 4.6.

---

[1] and simulations, if analytical expression is impossible for the performance metric of interest.

4.2 System Model and LS Schemes

4.2.1 System Model

The nodes of a distributed system are assumed to be 'homogeneous' in the sense that all nodes have the same arrival rate of external tasks[2] and are identical in processing capability and speed. Consequently, the task arrival/transfer activities experienced by each node are stochastically identical over a long term. Thus, we can adopt the general methodology — introduced in [ELZ86], and also used in [KC87, SC89a, LT86] — of first modeling the state (CET) evolution of a single node in isolation and then combining the node-level models into a system-level model. This decomposition was first verified (through simulation) in [ELZ86] to be valid for homogeneous systems of reasonably large size. We will also check its validity in Section 4.5 by comparing the analytic results with the results obtained from event-driven simulations.

External tasks (excluding transferred-in tasks) are assumed to arrive locally at node k according to a Poisson process with rate $\lambda_k = \lambda \; \forall k$. A task requires i units of time to execute and has j units of laxity time with probability q_{ij}, $1 \leq i \leq E_{max}$, $0 \leq j \leq L_{max}$, where E_{max} and L_{max} (measured in number of time units) are the largest task execution time and the largest task laxity in the system, respectively. $\{q_i = \sum_{j=0}^{L_{max}} q_{ij},\ 1 \leq i \leq E_{max}\}$ and $\{\hat{q}_j = \sum_{i=1}^{E_{max}} q_{ij},\ 0 \leq j \leq L_{max}\}$ are the probability distributions of task execution time and task laxity, respectively. All tasks are assumed to be independent of one another, so they do not communicate during their execution and thus have no precedence constraints among themselves. Note that aperiodic tasks in a real-time system are usually independent of each other. By contrast, periodic tasks often communicate with each other but their invocation, execution and communication behaviors are usually known *a priori* and thus scheduled off-line.

4.2.2 LS Schemes Under Consideration

We shall develop models for the proposed LS scheme as well as three other schemes: no LS, LS with random selection, and LS with perfect information. Besides, LS with state probing and LS with focused addressing [SRC85, RSZ89] will be comparatively assessed in Section 4.5 using simulations.

As was discussed in [ELZ86], a LS approach can be characterized by its transfer policy and its location policy. For analytical tractability, all LS schemes studied here em-

[2]These exclude transfer-in tasks.

Each node performs the following four operations:

(a): When a task with execution time T_i and laxity D_i arrives:
if current_time + CET $\geq D_i$ **then**
begin
receiver_node = table_lookup($\underline{x}$:observation)†;
transfer the task to receiver_node;
end
else
begin
CET := CET + T_i;
if CET crosses TH_{2k}, $1 \leq k \leq \lceil \frac{K_i}{2} \rceil - 1$ **then**
/* $TH_1, \ldots, TH_{K_i-1}$ are thresholds */
broadcast the state–region change to all nodes in its buddy set;
queue the task locally;
end

(b): When a message broadcast by node i, $1 \leq i \leq n$, arrives:
update observation x_i;
record the (observation, true state) pair needed for constructing probability distributions;

(c): **if** CET crosses TH_{2k}, $1 \leq k \leq \lceil \frac{K_i}{2} \rceil - 1$ **then**
broadcast the state–region change to all nodes in its buddy set;

(d): At every T_p clock ticks,
update the probability distributions and the table of loss–minimizing decisions;

†If a node anticipates, based on the current observation $\underline{x}$, that no other nodes can complete the task in time, this task is declared to be lost and thrown away.

Figure 4.1: Operations of the task scheduler on each node

ploy the transfer policy with the FCFS local scheduling discipline: a task with laxity ℓ is transferred from node i if and only if node i's CET is greater than ℓ. (and the operation of the task scheduler on each node in Fig. 3.1 is modified accordingly and is shown in Fig. 4.1). This transfer policy is of the threshold type as in [ELZ86, WM85, MTS89a, AC88], except that the threshold may change dynamically with the current state of the node and the time constraints of queued tasks. The location policies with which a node treats overflow tasks are different among LS schemes and are described in Section 3.5 in Chapter 3.

4.3 Analytic Models

Queueing models are developed to evaluate the performance of the proposed LS scheme as well as three other schemes: no LS, LS with random selection, LS with perfect information. We first model the state evolution of a node by a continuous time semi-Markov chain [Ros70, GH85] which will serve as the underlying model. The parameters of this model are derived for different LS schemes to characterize task arrival/transfer processes in the system level. A two–step iterative approach is then taken to obtain a numerical solution to the semi–Markov model.

4.3.1 The Underlying Model

The state of a node is defined as the CET of that node, and each node is modeled as an $M^{[x]}/D/1$ queue with bulk arrivals. (Arrival of a task with i units of execution time is viewed as the simultaneous arrival of i tasks, each requiring one unit of execution time.) The case in which all tasks require an identical execution time — and thus the state is QL — is a special case of this model.

The composite (both external and transferred) task arrival rate at a node is $\lambda(T)$, which depends on the node's CET, T, and the location/transfer policies used. Execution of a task requires i units of time with probability q_i, and such a task is called a *type–i task*, $1 \leq i \leq E_{max}$. Since at any time a node is either idle or busy executing a task, the node occupancy (by tasks) is divided into busy slots (measured in terms of system clock cycles) which are numbered as $B_1, B_2, \ldots$, relative to any reference point of time (see Fig. 4.2). Note that two adjacent busy slots of a node may be either contiguous or separated by idle periods. Let T_k denote a node's remaining CET at the end of B_k, and X_k^i represent the number of type–i arrivals during B_k, then

$$T_k = \begin{cases} T_{k-1} + \sum_{i=1}^{E_{max}} iX_k^i - 1, & \text{if } T_{k-1} > 0, \\ \sum_{i=1}^{E_{max}} iX_k^i + R - 1, & \text{if } T_{k-1} = 0, \end{cases} \tag{4.1}$$

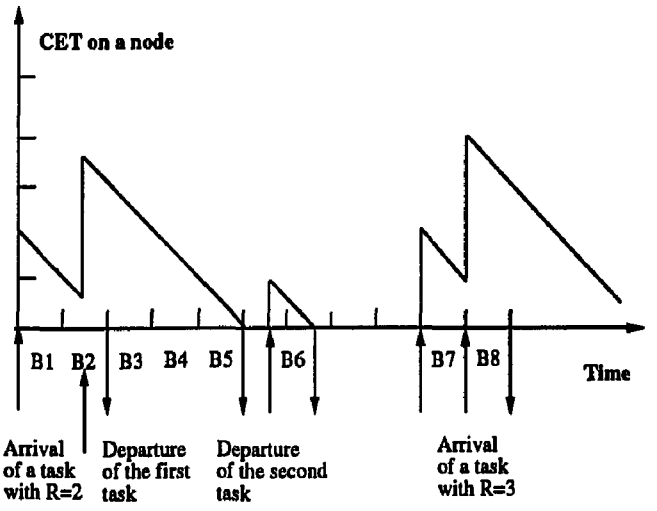


Figure 4.2: A sample path for the evolution of remaining CET on a node

where R is the number of clock cycles required for the node to complete a task which finds the node idle upon its arrival, and has the distribution $\{q_i, 1 \leq i \leq E_{max}\}$. If $T_{k-1} = 0$, then the node remains idle until a task with the required execution time, R, arrives. $\sum_{i=1}^{E_{max}} iX_k^i$ is the CET accrued during B_k with each type–i task contributing i units of time for execution. Eq. (4.1) describes a semi–Markov chain because (i) T_k depends only on T_{k-1}, and not on $T_{k'}\ \forall\ k' < k-1$, and (ii) the state residence times — the length of B_k — are deterministic with value 1, rather than exponentially–distributed.

Eq. (4.1) can be used to get the z–transform of the CET distribution. Let T^+ and T denote the CET on a node at some *embedding* time instant and at some *random* time instant, respectively. Because of the embedding points (i.e., at the end of each busy slot) chosen for the embedded Markov chain, the distribution of T, denoted by $p_T(.)$, is not necessarily the same as that of T^+, denoted by $p_T^+(.)$ [GH85, CT83]. However, $p_T(.)$ can be derived from $p_T^+(.)$ as described in [CT83]. So, let us derive $p_T^+(.)$ first.

Let $\Phi^+(z)$ denote the z–transform of CET distribution at the embedding time points, then

$$\Phi^+(z) = \sum_{n=0}^{\infty} p_T^+(n) z^n = E(z^{T^+}),$$

where $p_T^+(i) \triangleq \Pr(T^+ = i)$. Let $\delta(x)$ be the unit step function, then

$$T_k = T_{k-1} + \sum_{i=1}^{E_{max}} i \cdot X_k^i - 1 + (1 - \delta(T_{k-1})) \cdot R,$$

and

$$\begin{aligned}\Phi^{+}(z) &= E(z^{T^{+}}) = E(z^{T_k}) \\ &= E(z^{T_{k-1}+\sum_{i=1}^{E_{max}} iX_k^i+(1-\delta(T_{k-1}))R-1}) \\ &= E(z^{T_{k-1}+(1-\delta(T_{k-1}))R-1})E(z^{\sum_{i=1}^{E_{max}} iX_k^i}). \end{aligned} \tag{4.2}$$

Note that for mathematical tractability of Eq. (4.2), $\sum_{i=1}^{E_{max}} iX_k^i$ — the CET arrival process during B_k — is approximated to be independent of T_{k-1}, which is unrealistic for the location and transfer policies of the proposed LS scheme as well as others. As will be discussed in Section 4.3.2, this deficiency is remedied by figuring the dependency of the task arrival process on CET into task arrival rate. In other words, the task arrival rate λ on a node is determined by the node's CET, T, i.e., $\lambda(.)$ is a function of T.

The second factor of Eq. (4.2), $E(z^{\sum_{i=1}^{E_{max}} iX_k^i})$, is computed as follows. Since state residence times are all 1, we have

$$P(iX_k^i = in) = \frac{e^{q_i\lambda}(q_i\lambda)^n}{n!}$$

and

$$P(iX_k^i = \ell) = \begin{cases} \frac{e^{q_i\lambda}(q_i\lambda)^{\ell/i}}{(\ell/i)!} & \text{if } (\ell/i) \in N \cup \{0\} \\ 0 & \text{otherwise,} \end{cases}$$

where $1 \le i \le E_{max}$, and N is the set of natural numbers. In the above equation we used the fact that if each event of a Poisson process is classified independently of others to be any of type 1,2,...,E_{max} with probability q_i, then the number of type–i arrivals is independent of others, and is Poisson–distributed with λq_i. Let $U_k^i \triangleq iX_k^i$, then

$$\begin{aligned}\Phi_{U_k^i}(z) &= \sum_{\ell=0}^{\infty} z^{\ell} P(U_k^i = \ell) \\ &= \sum_{\ell=0}^{\infty} z^{i\ell} P(U_k^i = i\ell) \\ &= \sum_{\ell=0}^{\infty} z^{i\ell} \cdot \frac{e^{q_i\lambda}(q_i\lambda)^{\ell}}{\ell!} \\ &= e^{q_i\lambda(z^i-1)}. \end{aligned} \tag{4.3}$$

Assuming independent arrivals of different types of tasks, we then have

$$\begin{aligned}\Phi_{\sum_{i=1}^{E_{max}} U_k^i}(z) &= \prod_{i=1}^{E_{max}} \Phi_{U_k^i}(z) = \prod_{i=1}^{E_{max}} e^{q_i\lambda(z^i-1)} \\ &= e^{\lambda(\sum_{i=1}^{E_{max}} q_i z^i - 1)} = e^{\lambda(\Phi_R(z)-1)}, \end{aligned} \tag{4.4}$$

where $\Phi_R(z)$ is the z–transform of the required task execution time.

We now compute the first factor of Eq. (4.2). The event, $\{T_{k-1}-1+(1-\delta(T_{k-1}))\cdot R=i\}$, contains two mutually exclusive subevents: (1) $T_{k-1}=0$ and $R=i+1$, and (2) $T_{k-1}=i+1$:

$$\Pr(T_{k-1}-1+(1-\delta(T_{k-1}))\cdot R=i) \;=\; q_{i+1}\cdot p_T^+(0)+p_T^+(i+1),$$

and thus,

$$\begin{aligned} E(z^{T_{k-1}-1+(1-\delta(T_{k-1}))R}) &= \sum_{n=0}^{\infty}\Pr(T_{k-1}-1+(1-\delta(T_{k-1}))R=n)\cdot z^n \\ &= \sum_{n=0}^{\infty}(q_{n+1}\cdot p_T^+(0)+p_T^+(n+1))\cdot z^n \\ &= \frac{1}{z}p_T^+(0)\Phi_R(z)+\frac{1}{z}(\Phi(z)-p_T^+(0)). \end{aligned} \tag{4.5}$$

Substituting Eqs. (4.4) and (4.5) into Eq. (4.2), and rearranging the terms, we get:

$$\Phi^+(z)=\frac{\frac{1}{z}p_T^+(0)(\Phi_R(z)-1)e^{\lambda(\Phi_R(z)-1)}}{1-\frac{1}{z}e^{\lambda(\Phi_R(z)-1)}}. \tag{4.6}$$

$p_T^+(0)$ can be obtained from:

$$\begin{aligned} 1 &= \Phi^+(1) \\ &= p_T^+(0)\cdot\lim_{z\to 1}\frac{\frac{1}{z}(\Phi_R(z)-1)e^{\lambda(\Phi_R(z)-1)}}{1-(1/z)e^{\lambda(\Phi_R(z)-1)}} \\ &= p_T^+(0)\cdot\frac{E(R)}{1-\lambda E(R)}, \end{aligned}$$

where we have used L'Hopital's rule in evaluating the limit. Consequently,

$$\begin{aligned} p_T^+(0) &= \frac{1-\lambda E(R)}{E(R)} \\ &= (1/E(R))-\lambda, \end{aligned} \tag{4.7}$$

where $E(R)$ is the expected execution time, i.e., $E(R)=\sum_{i=1}^{E_{max}} iq_i$. Note that for the system to be stable (or for the CET at a node not to grow unboundedly), we must have $\sum_{i=1}^{E_{max}} i\lambda q_i \le 1$, or $\lambda \le (1/E(R))$, which is the necessary and sufficient condition for $p_T^+(0)\ge 0$. Thus,

$$\Phi^+(z)=\frac{(\frac{1}{E(R)}-\lambda)(\Phi_R(z)-1)e^{\lambda(\Phi_R(z)-1)}}{z-e^{\lambda(\Phi_R(z)-1)}}. \tag{4.8}$$

The above results for the embedded Markov chain do not directly apply to the total general–time stochastic process. However, the relation between the general–time steady–state distribution, $p_T(.)$, and the distribution at embedding points, $p_T^+(.)$, is shown in [CT83] to be

$$\Phi(z)=\frac{E(R)(z-1)}{\Phi_R(z)-1}\cdot\Phi^+(z), \tag{4.9}$$

for $M^{[x]}/G/1$ systems, where $\Phi(z)$ is the z–transform of the general–time CET distribution. Thus, we have

$$\Phi(z) = \frac{(1-\lambda E(R))(z-1)e^{\lambda(\Phi_R(z)-1)}}{z - e^{\lambda(\Phi_R(z)-1)}}, \tag{4.10}$$

and

$$p_T(0) = 1 - \lambda E(R). \tag{4.11}$$

If all tasks require an identical execution time, i.e., $\Pr(R = 1) = 1$, then the state reduces to QL, and Eq. (4.11) reduces to

$$p_T(0) = 1 - \lambda,$$

which is exactly the utilization $U = \lambda$ since the service time is 1, and Eq. (4.10) (and also Eq. (4.8)) reduces to

$$\Phi(z) = \frac{(1-\lambda)(z-1)e^{\lambda(z-1)}}{z - e^{\lambda(z-1)}}. \tag{4.12}$$

Computing the inverse z–transform of Eq. (4.10) (Eq. (4.12)) numerically yields the CET (QL) distribution, $\{p_T(i), i \geq 0\}$. The discussion of a rather subtle technique for the inversion of Eq. (4.10) can be found in [CT83].

4.3.2 Derivation of $\lambda(T)$

The semi–Markov chain model derived above can be used to evaluate different LS schemes if $\lambda(T)$ characterizes both the corresponding task arrivals and/or task transfers in the system level. The following variables are necessary to facilitate the derivation of $\lambda(T)$:

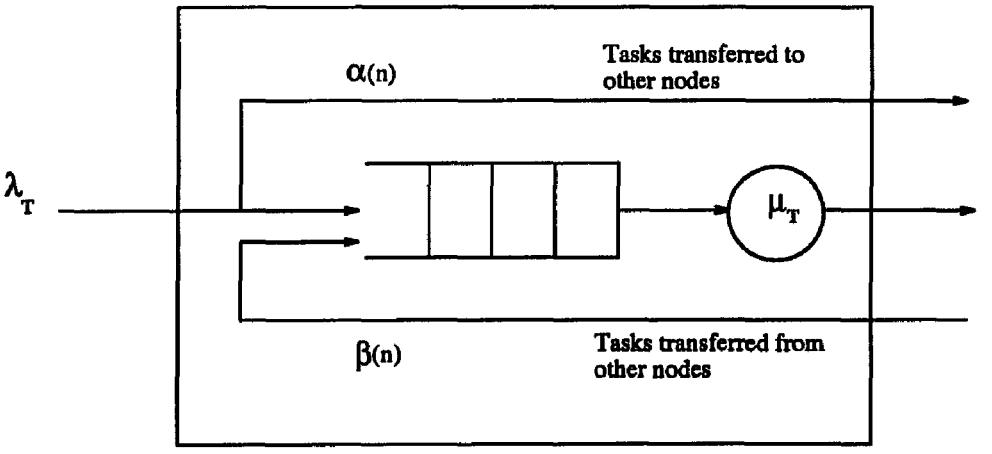
- $\alpha(n)$: the rate of transferring tasks out of a node given that the node's remaining CET is T. Since the transfer policy determines whether or not a task can be completed in time locally, this parameter characterizes the transfer policy used.

- $\beta(T)$: the rate of transferring tasks *into* a node given that the node's remaining CET is T. This parameter corresponds to the location policy used, since the location policy determines where to send each overflow task.

- γ_j: the probability that the remaining CET on a node is no less than j units of time, i.e., $\gamma_j = \Pr(T \geq j)$.

- K_j: the number of nodes that can be chosen by a node, excluding itself, for transferring a task with laxity j. The distribution of K_j can be expressed in terms of γ_j as:

$$\Pr(K_j = \ell) = \binom{n-1}{\ell} (1-\gamma_{j+1})^{\ell} \gamma_{j+1}^{n-1-\ell}.$$



where $\alpha(n)$ is the rate of transferring tasks out of a node given that N=n

$\beta(n)$ is the rate of transferring tasks into a node given that N=n

and $\lambda(n) = \lambda_T - \alpha(n) + \beta(n)$

Figure 4.3: A generic queueing model for each node.

As shown in Fig. 4.3, $\lambda(T) = \lambda - \alpha(T) + \beta(T)$. By appropriately tailoring $\alpha(T)$ and $\beta(T)$ to describe the transfer and location policies adopted and by approximating the combined (external and transferred-in) task arrivals at each node to follow a Poisson process, the above semi-Markov chain model can be used to express the operations of different LS schemes. This approximation is accurate only when (1) task-transfer out of each node is a Poisson process, implying that the decision on whether or not to transfer a task is independent of the current workload [Kle75], which is not true for our LS scheme, and (2) task-transfer into each node — the superposition of task transfers out of other nodes — is Poisson. However, this approximation is verified by our simulation experiments to be valid for light to medium loaded distributed systems of size ≥ 12. For example, when the task transfer-out ratio is less than 40% of the task arrival rate, the combined task arrival processes at each node have the coefficients of variation of their interarrival times close to one.[3] A statistical approach to validate this assumption will be treated in Section 5.5.1.

Moreover, for all LS schemes the following relationship between $\alpha(T)$ and $\beta(T)$ results from the law of task conservation, $\sum_{k=0}^{\infty} \lambda(k) \cdot p_T(k) = \lambda$.

Theorem 1 *If task flow of the system is conserved, then*

$$\sum_{k=0}^{\infty} \alpha(k) \cdot p_T(k) = \sum_{k=1}^{\infty} \beta(k) \cdot p_T(k). \tag{4.13}$$

[3]Which is at least a necessary condition for the combined task arrival processes to be modeled as Poisson.

The transfer policy used in real–time systems is of the dynamic threshold type: a task is transferred to other nodes only if it cannot be completed in time locally. Thus, given a node's remaining CET = T, all the tasks arrived with laxities smaller than T should be transferred, thus leading to:

$$\alpha(T) = \sum_{j=0}^{T-1} \lambda \hat{q}_j, \tag{4.14}$$

for all schemes except for no LS in which $\alpha(T) = \beta(T) = 0$, and $\lambda(T) = \lambda \ \forall T$.

Unlike the transfer policy, the location policy depends on each LS scheme. The random selection scheme selects randomly a receiver node in the system for each overflow task without using any state information. Thus, we get $\beta(T)$ for this scheme as[4]

$$\beta(T) = \frac{1}{N} \sum_{j=1}^{L_{max}} (N\gamma_{j+1}) \cdot \lambda \hat{q}_j = \sum_{j=1}^{L_{max}} \gamma_{j+1} \lambda \hat{q}_j, \tag{4.15}$$

where $N\gamma_{j+1}$ is the average number of nodes that cannot complete tasks with laxity j in time in an N–node system, $\lambda \hat{q}_j$ is the arrival rate of tasks with laxity j on a node, and the product of these two is the average rate of transferring tasks with laxity j in the system. Since all overflow tasks are transferred randomly, each node shares $1/N$ of these tasks. The randomness property is reflected in the independence of $\beta(T)$ from T. The correctness of Eq. (4.15) is verified in the Appendix A.

Our proposed LS scheme uses the posterior distributions derived from the state information gathered from time–stamped region–change broadcasts to estimate the workload of other nodes, and chooses probabilistically, from the preferred list, the best candidate to which each overflow task will be transferred. If these distributions are properly constructed, then $\beta(T)$ can be expressed as:

$$\begin{aligned} \beta(T) &= \sum_{j=T}^{L_{max}} \lambda \hat{q}_j \gamma_{j+1} \cdot (1 + \gamma_{j+1} + \gamma_{j+1}^2 + \ldots + \gamma_{j+1}^{N_B - 1}) \\ &= \sum_{j=T}^{L_{max}} \lambda \hat{q}_j \gamma_{j+1} \cdot \frac{1 - \gamma_{j+1}^{N_B}}{1 - \gamma_{j+1}}, \end{aligned} \tag{4.16}$$

where N_B is the number of nodes in a buddy set. Note that (1) a node i with CET = T can complete all tasks with laxity greater than T in time, and thus, the summation is performed from T to L_{max}, and (2) the term $\lambda \hat{q}_j \gamma_{j+1}$ is contributed by the node whose most preferred node is node i, the term $\lambda \hat{q}_j \gamma_{j+1}^2$ is contributed by the node whose second preferred node is node i and whose most preferred node cannot complete tasks with laxity j in time, and

[4] $\beta(T)$ is derived under the assumption that the task–transfer into a node — the superposition of task–transfers out of other nodes — is Poisson–distributed.

the term $\lambda\hat{q}_j\gamma_{j+1}^{N_B}$ accounts for the situation when all nodes in a buddy set cannot complete tasks with laxity j in time. The correctness of Eq. (4.16) is verified in the Appendix A.

As discussed in Chapter 3, the preferred list should be constructed so that **(P1)** and **(P2)** are satisfied. These properties minimize the possibility of multiple nodes simultaneously sending tasks to the same 'capable' node, while ensuring overflow tasks to be evenly shared by 'capable' nodes. More formally, we have the following theorem:

Theorem 2 *Using the preferred lists and proper prior/posterior distributions, our proposed scheme balances load in the sense that all overflow tasks are evenly shared by those capable nodes.*

Proof: This theorem is proved by deriving $\beta(T)$ based on the idea of even sharing of overflow tasks among 'capable' nodes and comparing the result with the $\beta(T)$ in Eq. (4.16). Even sharing of overflow tasks gives:

$$\beta(T) = \sum_{j=T}^{L_{max}} \sum_{\ell=0}^{N_B-1} \frac{(N_B\gamma_{j+1})\cdot(\lambda\hat{q}_j)}{\ell+1} \cdot \Pr(K_j = \ell)$$

where $\lambda\hat{q}_j$ is the arrival rate of tasks with laxity j, $N_B\gamma_{j+1}$ is the average number of nodes which cannot complete tasks with laxity j in time. The product of these two is the average rate of transferring tasks with laxity j, which will be shared evenly by ℓ other nodes (in addition to the node itself) with probability $\Pr(K_j = \ell)$. $\beta(T)$ can be simplified to:

$$\begin{aligned}\beta(T) &= \sum_{j=T}^{L_{max}} \lambda\hat{q}_j\gamma_{j+1} \sum_{\ell=0}^{N_B-1} \frac{N_B}{\ell+1} \Pr(K_j = \ell) \\ &= \sum_{j=T}^{L_{max}} \lambda\hat{q}_j\gamma_{j+1} \cdot \frac{1-\gamma_{j+1}^{N_B}}{1-\gamma_{j+1}} \qquad (4.17)\end{aligned}$$

which is exactly the same as Eq. (4.16). □

The location policy of the quasi–perfect LS scheme is similar to that of our scheme except that (i) accurate state information is obtained without incurring any communication cost, (ii) there is no overhead associated with task transfers, and (iii) the buddy set size N_B equals the number of processing nodes, K_{pn}, in the distributed system. That is, overflow tasks are transferred directly and instantaneously to 'capable' nodes. $\lambda(T)$ can be expressed as:

$$\begin{aligned}\lambda(T) &= \sum_{j=T}^{L_{max}} \sum_{\ell=0}^{K_{pn}-1} \frac{K_{pn}\lambda\hat{q}_j}{\ell+1} \cdot \Pr(K_j = \ell) \\ &= \sum_{j=T}^{L_{max}} \lambda\hat{q}_j \cdot \frac{1-\gamma_{j+1}^{K_{pn}}}{1-\gamma_{j+1}}. \qquad (4.18)\end{aligned}$$

The correctness of Eq. (4.18) is also verified in the Appendix A.

4.3.3 An Iterative Algorithm

$\lambda(T)$ ($\alpha(T)$ and $\beta(T)$) must be known before solving the Markov chain model for $p_T(.)$. However, $\lambda(T)$ depends on γ_j which in turn depends on $p_T(.)$. An iterative approach is taken to handle the difficulty associated with this recursion problem. Note that $\Phi_{\sum_{i=1}^{E_{max}} U_k^i}(z) = e^{\lambda(\Phi_R(z)-1)}$ (Eq. (4.4)) can be interpreted as the pdf of the number of arrivals[5] during one unit of service time (execution time). Thus, we modify Eq. (4.4) as

$$\Phi_{\sum_{i=1}^{E_{max}} U_k^i}(z) = \sum_{i=0}^{\infty} e^{\lambda(i)(\Phi_R(z)-1)} p_T(i),$$

to account for the effect that the task arrival rate varies with the current CET, T, of a node. Consequently, Eq. (4.10) is modified as

$$\Phi(z) = \frac{(1-\lambda E(R))(z-1)\sum_{i=0}^{\infty} e^{\lambda(i)(\Phi_R(z)-1)} p_T(i)}{z - \sum_{i=0}^{\infty} e^{\lambda(i)(\Phi_R(z)-1)} p_T(i)},$$

and Eq. (4.12) as

$$\Phi(z) = \frac{(1-\lambda)(z-1)\sum_{i=0}^{\infty} e^{\lambda(i)(z-1)} p_T(i)}{z - \sum_{i=0}^{\infty} e^{\lambda(i)(z-1)} p_T(i)}.$$

In the first step, the modified version of Eq. (4.10) (Eq. (4.12)) is solved for $p_T(.)$ with both $\alpha(T)$ and $\beta(T)$ set to 0, or equivalently, $\lambda(T) = \lambda \ \forall T$. The resulting $p_T(.)$ is used to compute $\alpha(T)$ and $\beta(T)$ in the second step. Then $p_T(.)$ is recalculated with the new $\alpha(T)$ and $\beta(T)$ (and thus a new $\lambda(T)$) using the modified version of Eq. (4.4) and Eq. (4.10). This result will, in turn, change $\alpha(T)$ and $\beta(T)$. This procedure will repeat until $p_T(.)$ and $\lambda(T)$ converge to some fixed values.

4.4 Computation/Communication Overheads

To develop a practical model for assessing the performance of different LS schemes, one should, in addition to addressing the fundamental differences among the LS schemes, take into account their implementation overheads: for example, the computational overheads of our scheme due to probability updates, and communication delays associated with state–information collection and task transfers. The tradeoff between the associated complexity and the resulting benefit can be analyzed accurately only if implementation overheads are included in the model. In this section, we extend our model to include the overheads of the proposed scheme due to time–stamped region–change broadcasts, periodic updates of posterior distributions, and task transfers by:

---

[5] As mentioned earlier, a type–i task arrival is viewed as i simultaneous arrivals, each with one unit of execution time.

- Augmenting the original task set with a new type of tasks, i.e., the probability updating tasks.
- Modifying the underlying semi–Markov chain model to include the effect of region–change broadcasts on the CET of a node.
- Considering the effect of communication delay by modifying $\alpha(T)$ and $\beta(T)$.

4.4.1 Overheads of Probability Updates

As mentioned in Chapter 3, each node updates the posterior distributions of other nodes' CET once every T_p units of time. Let C_p be the time required for updating probability distributions, then we introduce a new type of task by modifying the parameters λ and q_{ij} (which characterize the task set) as:

$$\lambda' = \lambda + \frac{1}{T_p}, \quad (\lambda + \frac{1}{T_p}) \cdot q'_{C_p L_{max}} = \frac{1}{T_p},$$

and

$$(\lambda + \frac{1}{T_p}) \cdot q'_{ij} = \lambda q_{ij},$$

where q'_{ij} and q_{ij} (λ' and λ) are the new and old values of the probabilities (task arrival rates), respectively. That is, a new type of tasks is added to the task set: the one with execution time C_p and laxity L_{max}. Moreover, q_{ij}'s are scaled in accordance with the new task arrival rate λ'. Note that (1) the laxity of the probability updating task is chosen to be L_{max}, since this task is not time–critical, and (2) for ease of analysis, this periodic task is modeled to have exponential time–to–event distributions with the rates equal to the reciprocal of the period. The error of this approximation relative to the exact limit–equivalent rates is bounded as indicated by Kitchin [Kit88].

4.4.2 Overheads of Region–Change Broadcasts

Recall that each node broadcasts the change of state region to all the other nodes in its buddy set. Let C_b be the time needed for broadcasting a region–change. Since this broadcasting process is state–dependent, the overheads of region–change broadcasts can be included by modifying Eq. (4.1), the expression for state evolution as:

$$T_k = \begin{cases} T_{k-1} + \sum_{i=1}^{m} iX_k^i - 1 + C_b & \text{if } T_{k-1} > 0 \text{ and } T_{k-1} \in \{TH_{2\ell},\ 1 \le \ell \le \lceil \frac{K_1}{2} \rceil - 1\} \\ T_{k-1} + \sum_{i=1}^{m} iX_k^i - 1 & \text{if } T_{k-1} > 0 \text{ and } T_{k-1} \notin \{TH_{2\ell},\ 1 \le \ell \le \lceil \frac{K_1}{2} \rceil - 1\} \\ \sum_{i=1}^{m} iX_k^i + R - 1 & \text{if } T_{k-1} = 0. \end{cases} \tag{4.19}$$

In other words, whenever the remaining CET reaches TH_{2k} ($1 \leq k \leq \lceil \frac{K_i}{2} \rceil - 1$), C_b units of time are added to the state to account for the CET increase due to broadcasts. The property of semi-Markov chain is retained, because whether or not to increase by C_b units of time depends only on T_{k-1}. Similarly, Eq. (4.2) should be rewritten as:

$$\Phi^+(z) = E(z^{T_{k-1}+(1-\delta(T_{k-1}))R + C_b(\sum_{\ell=1}^{\lceil \frac{K_i}{2} \rceil - 1} \delta'(T_{k-1}-TH_{2\ell}))-1}) E(z^{\sum_{i=1}^{m} iX_k^i})$$

where $\delta'(x)$ is the impulse function or the derivative of the unit step function, $\delta(x)$. Following the same (but more complex) derivation as in Section 4.3, one can get a modified version of Eq. (4.10):

$$\begin{aligned} \Phi(z) &= \frac{(1 - \lambda E(R) - \sum_{\ell=1}^{\lceil \frac{K_i}{2} \rceil - 1} p_T(TH_{2\ell}))(z-1)e^{\lambda(\Phi_R(z)-1)}}{z - e^{\lambda(\Phi_R(z)-1)}} \\ &+ \frac{e^{\lambda(\Phi_R(z)-1)}(z^{C_b}-1)(\sum_{\ell=1}^{\lceil \frac{K_i}{2} \rceil - 1} p_T(TH_{2\ell}) z^{TH_{2\ell}-1})}{1 - \frac{1}{z}e^{\lambda(\Phi_R(z)-1)}} \cdot \frac{E(R)(z-1)}{\Phi_R(z)-1}. \end{aligned} \quad (4.20)$$

Note that the above equations reduce to Eq. (4.10) if $C_b = 0$ (no broadcasting overhead). Before solving Eq. (4.20) for $p_T(.)$, one has to know the values of $p_T(TH_{2\ell})$, $1 \leq \ell \leq \lceil \frac{K_i}{2} \rceil - 1$, which, in turn, depend on $\Phi(z)$. This recursive dependency can again be handled by using an iterative approach as follows. $\Phi(z)$ is first inverse z-transformed with $p_T(TH_{2\ell})$ set to 0. The resulting $p_T(.)$ (in particular, $p_T(TH_{2\ell})$'s) is then used to compute the new $\Phi(z)$ from which new $p_T(.)$ can then be calculated. This process will be repeated until $p_T(.)$ converges to a fixed value.

4.4.3 Effect of Communication Delays

Communication delays are composed of three components [BG87]: (1) the queueing delay, which is the time between the queueing of a task and the start of its transfer, (2) the transmission delay, which is the time between the first and last bits of the task transferred, and (3) the propagation delay, which is the time from the transmission of a bit at the sending node to its receipt by the destination node. The transmission delay depends on the size of the transferred task, and is thus assumed to be proportional (with ratio o_1) to the required execution time of the task. The propagation delay depends on the physical distance/characteristics of the communication medium between the sender and receiver nodes, is independent of traffic loads, and is thus assumed to be constant, o_2. The queueing delay depends heavily on traffic loads. Since region-change broadcasts and task transfers introduce additional traffic loads, the queueing delay under the proposed scheme is expected to be larger than others. However, the exact dependency of the queueing delay on these operations is difficult to model, because (1) the delay also depends on the capacity of the

communication medium and the (contention) protocols used, both of which are application–dependent, and (2) the effect of region–change broadcasts on this delay depends on the state of the system, which changes dynamically with time. We thus assume in our model that the queueing delay due to task transfers and region–change broadcasts are proportional to (with ratio o_3 and o_4) the task transfer–out ratio (r_{tt}) and external task arrival rate (λ), respectively[6]. Let $c_R(i)$ and $c_P(i)$ (r_{tt}^R and r_{tt}^P) denote the communication overheads (task transfer–out ratios) encountered by a task with i units of execution time in the random selection scheme and the proposed scheme, respectively, then we have $c_R(i) = o_1 \cdot i + o_2 + o_3 \cdot r_{tt}^R$ and $c_P(i) = o_1 \cdot i + o_2 + o_3 \cdot r_{tt}^P + o_4 \cdot \lambda$.

Considering the effect of communication delays, $\alpha(T)$ in Eq. (4.14) should be modified as:

$$\alpha(T) = \sum_{i=1}^{E_{max}} \sum_{j=\lceil c(i) \rceil}^{T-1} \lambda q_{ij}, \tag{4.21}$$

where $c(i)$ is $c_P(i)$ or $c_R(i)$, depending the scheme under consideration. Note that for those tasks whose laxity is less than the communication delay, there is no need to transfer them. Similarly, $\beta(T)$ is modified as:

$$\beta(T) = \sum_{i=1}^{E_{max}} \sum_{j=\lceil c_R(i) \rceil}^{L_{max}} \lambda \gamma_{j+1} q_{ij} \tag{4.22}$$

for the random selection scheme, and

$$\beta(T) = \sum_{i=1}^{E_{max}} \sum_{j=\lceil c_P(i) \rceil}^{L_{max}} \lambda q_{ij} \gamma_{j+1} \cdot \frac{1 - \gamma_{j+1}^{N_B}}{1 - \gamma_{j+1}} \tag{4.23}$$

for the proposed LS scheme. Correctness of these expressions can be verified similarly to Corollaries shown in the Appendix A.

4.5 Performance Analysis

To analytically assess the proposed LS scheme and to validate the analytic models, we present numerical results for the case when inter–arrival times of external tasks are exponentially distributed. (Note, however, that the proposed LS scheme is not restricted to exponential distributions.) The proposed scheme and four other LS schemes, i.e., no LS, LS with random selection, LS with state probing,[7] and quasi–perfect LS, are comparatively evaluated with analytic models.

---

[6]The reasons for assuming this linear relationship is that we can easily compute these coefficients using linear prediction and the data obtained from simulations.

[7]Only simulation results are shown for this scheme in our analysis.

The system configuration, the size of the buddy set, the maximum number of nodes probed for each overflow task, the tunable parameters chosen in the proposed LS scheme, the computational overheads assumed, the task parameter ranges varied, the confidence level achieved (in simulations), and the notation used all conform to those described in Section 3.5. The transmission delay associated with each task transfer is assumed to be 10 % of $E(R)$, i.e., $o_1 = 0.1$. The propagation delay is assumed to be 1 % of $E(R)$, i.e., $o_2 = 0.01$. The coefficients associated with the queueing delay due to task transfers (o_3), region–change broadcasts (o_4), and state probes (o_5) are set to 0.1, 0.05, and 0.01, respectively.[8]

We validate our analytical models by comparing the numerical results obtained from analytical models against those obtained from simulations. We also evaluate different LS schemes with respect to several important performance metrics which are derived from the analytic models, i.e., the distribution of CET, the probability of dynamic failure, the mean response time, and the task transfer–out ratio. The performance with respect to metrics that cannot be directly derived from analytical models has been evaluated using simulations in Section 3.5.3 in Chapter 3.

4.5.1 Evaluation of Important Performance Metrics

Distribution of CET

The CET distribution, $p_T(\cdot)$, can be obtained by using either Eq. (4.10) or Eq. (4.20), depending on the scheme under consideration. Table 4.1 gives some numerical examples of the CET distribution with respect to different distributions of task laxity for different LS schemes. The CET distribution obtained via simulation is shown to be very close to the analytic solution, with a 5 % error in the cumulative distribution, indicating the validity of the analytic models. So, we shall henceforth use the numerical results derived from the analytic models in the subsequent discussion, unless stated otherwise.

The CET distributions with respect to different distributions of task execution time are plotted in Fig. 4.4 and Fig. 4.5 with $\lambda = 0.8$. The numerical results are so close to one another among the state probing scheme, the random selection scheme, and the proposed scheme that only one curve corresponding to the proposed scheme is plotted. (Also, the results for no LS obtained from analytic modeling and simulations are so close to each other that they are almost indistinguishable in Figs. 4.4–4.5.) The CET distributions under different LS schemes approach unity much faster than those without LS, thus justifying

[8] All overhead-related parameters are based on the data obtained from simulations.

$(\lambda = 0.8)$		No LS		State prob.	Random selection		Proposed scheme		Perfect scheme	
Task attributes	$CET \leq$	anal.	simul.	simul.	anal.	simul.	anal.	simu.	anal.	simul.
	0.0	0.2000	0.2138	0.2133	0.2121	0.2212	0.2609	0.2568	0.2914	0.2810
	0.8	0.3867	0.3710	0.5995	0.6216	0.6202	0.6920	0.6911	0.6474	0.6421
	1.6	0.5281	0.5246	0.8547	0.8602	0.8653	0.9019	0.9080	0.8858	0.8897
$ET = \{0.4, 0.8,$	2.4	0.6316	0.6362	0.9642	0.9611	0.9665	0.9811	0.9840	0.9820	0.9867
$1.2, 1.6\}_{0.25},$	3.2	0.7263	0.7222	0.9929	0.9887	0.9933	0.9975	0.9987	0.9998	0.9998
$L = \{1, 2, 3\}_{1/3}$	4.0	0.7816	0.7874	0.9981	0.9961	0.9979	0.9982	0.9995	1.0000	1.0000
	4.8	0.8412	0.8369	0.9996	0.9990	0.9997	0.9986	0.9998	1.0000	1.0000
	5.6	0.8824	0.8751	0.9999	0.9998	1.0000	0.9992	0.9999	1.0000	1.0000
	6.4	0.9086	0.9044	1.0000	1.0000	1.0000	0.9998	0.9999	1.0000	1.0000
	7.2	0.9293	0.9259	1.0000	1.0000	1.0000	0.9999	0.9999	1.0000	1.0000
	0.0	0.2000	0.2153	0.2094	0.2192	0.2287	0.2683	0.2853	0.3428	0.3787
	0.8	0.3867	0.3749	0.6473	0.7380	0.7210	0.8614	0.8409	0.8612	0.8469
	1.6	0.5281	0.5284	0.8890	0.9184	0.9262	0.9686	0.9770	0.9879	0.9961
$ET = \{0.4, 0.8,$	2.4	0.6316	0.6406	0.9581	0.9662	0.9741	0.9860	0.9910	0.9913	0.9991
$1.2, 1.6\}_{0.25},$	3.2	0.7263	0.7287	0.9857	0.9892	0.9930	0.9912	0.9957	0.9948	0.9997
$L = \{0.4, 0.8,$	4.0	0.7816	0.7902	0.9949	0.9923	0.9980	0.9930	0.9977	0.9968	0.9999
$1.2, 1.6\}_{0.25},$	4.8	0.8412	0.8419	0.9980	0.9961	0.9995	0.9948	0.9988	0.9984	1.0000
	5.6	0.8824	0.8803	0.9992	0.9984	0.9998	0.9978	0.9992	0.9996	1.0000
	6.4	0.9086	0.9128	0.9997	0.9996	1.0000	0.9984	0.9995	1.0000	1.0000
	7.2	0.9293	0.9301	0.9998	0.9999	1.0000	0.9990	0.9996	1.0000	1.0000

Table 4.1: CET distributions for different task sets under different schemes ($N = 16$).

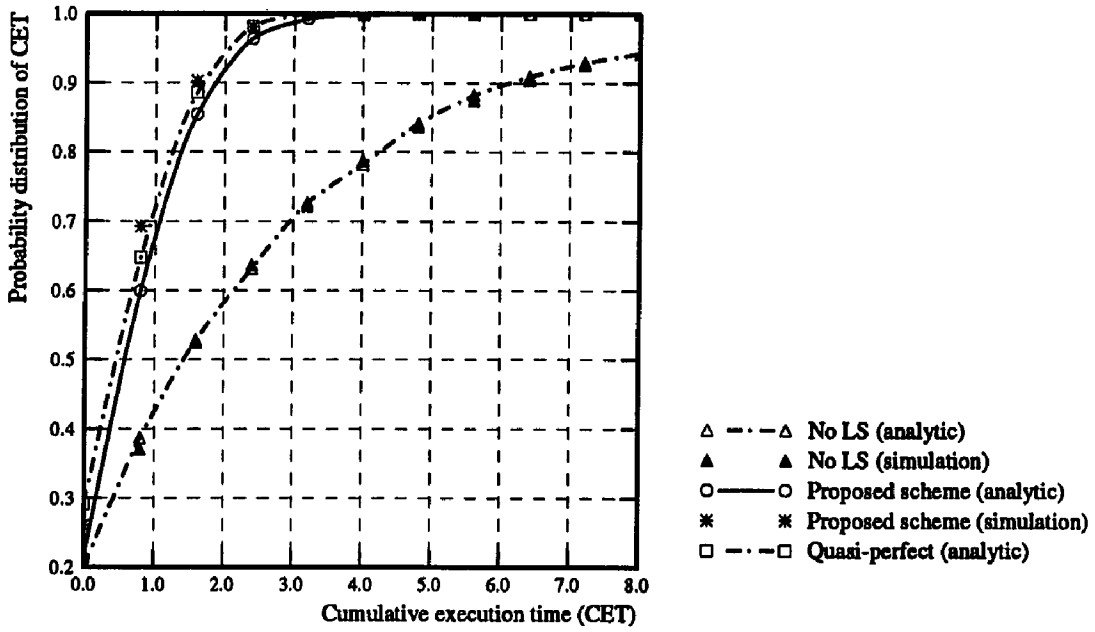


Figure 4.4: Probability distribution of CET for a task set with $\lambda = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and $L = \{1, 2, 3\}_{1/3}$.

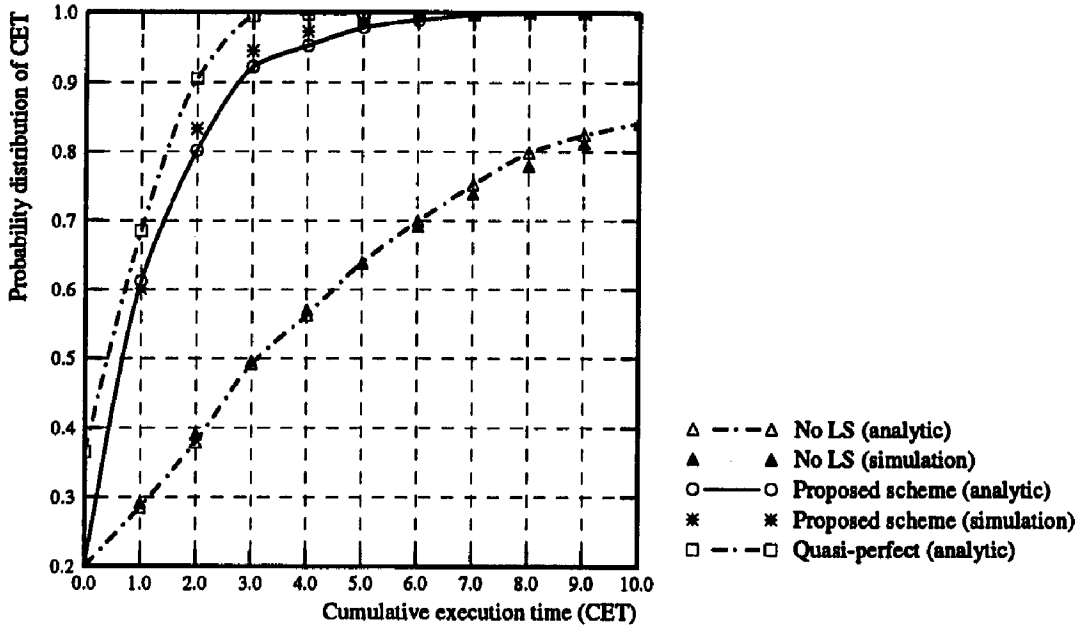


Figure 4.5: Probability distribution of CET for a task set with $\lambda = 0.8$, $ET = \{0.027, 0.27, 2.7\}_{1/3}$, and $L = \{1, 2, 3\}_{1/3}$.

the need of LS to handle bursty task arrivals in distributed systems. Besides, the CET distributions vary significantly as the distribution of task execution time varies; QL is thus not adequate to measure the workload of a node (as was verified in Section 3.5.3).

One interesting result is that the CET distribution associated with the proposed scheme does not approach unity with the fastest speed (Table 4.1). However, the proposed LS scheme does have a higher $P(T \leq t)$ than others for $\forall\ t \in (0, L_{max}]$. This is because the proposed scheme, instead of trying to minimizing the average CET on each node, aims to make each node's CET less than the laxity of any arrived task, so that P_{dyn} can be minimized.

Another interesting result is that CET distributions vary with the distributions of task laxity even when the distributions of task execution time are the same. This is because of the real–time application-oriented transfer policy used, where the laxity of an incoming task, rather than certain thresholds as in [ELZ86, SC89a, PTS88], is used to determine whether or not to transfer a task. Consequently, we try to minimize P_{dyn} with respect to the distributions of *both* task laxity and task execution time, instead of balancing load *only* with respect to the distribution of task execution time.

Probability of Dynamic Failure

As discussed in Section 3.5.3, a dynamic failure occurs if the sum of the queueing-for-execution time and the task–transfer (if any) time exceeds the laxity of a task. Let $P_{dyn|\ell,e}$, $P_{dyn|\ell}$, and P_{dyn} denote the probability of missing the deadline of a task with execution time e and laxity ℓ, the probability of missing the deadline of a task with laxity ℓ, and the probability of dynamic failure, respectively. Then, $P_{dyn} = \sum_{j=0}^{L_{max}} P_{dyn|j}\hat{q}_j$, and, according to our queueing model, the other two probabilities can be expressed in terms of γ_j and $\hat{q}_j$ as:

1. Non–cooperative scheme (no LS): $P_{dyn|\ell,e} = P_{dyn|\ell} = \gamma_{\ell+1}$, where γ_j is calculated from Eq. (4.10) with $\lambda(T) = \lambda\ \forall T$.

2. LS scheme with random selection:

$$P_{dyn|\ell,e} = \prod_{i=0}^{n_\ell^e} \gamma_{(\ell+1-ic_R(e))}, \quad \text{and} \quad P_{dyn|\ell} = \sum_{i=1}^{E_{max}} P_{dyn|\ell,i}\, q_i,$$

where $c_R(e)$ is the communication overhead associated with a task with execution time e under the random selection scheme, $n_\ell^e = \lfloor \frac{\ell}{c_R(e)} \rfloor$, and γ_j is obtained from Eq. (4.10) using $\alpha(T)$ in Eq. (4.21) and $\beta(T)$ in Eq. (4.22).

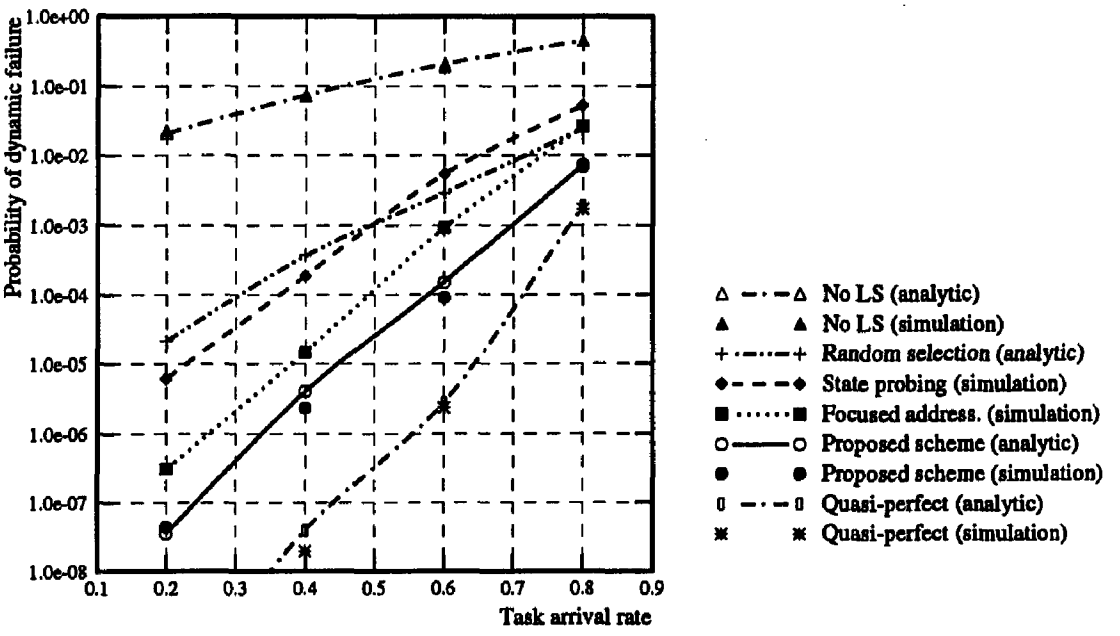


Figure 4.6: P_{dyn} vs. task arrival rate for a 16-node system with a task set: $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{1/3}$.

3. Perfect information scheme: $P_{dyn|\ell,e} = P_{dyn|\ell} = \gamma_{\ell+1}^{K_{pn}}$, where γ_j is calculated from Eq. (4.10) using $\lambda(T)$ in Eq. (4.18), and K_{pn} is the number of processing nodes in the distributed system.

4. The proposed scheme:

$$P_{dyn|\ell,e} = \prod_{i=0}^{n_\ell^e} \gamma_{(\ell+1-ic_P(e))},$$

and

$$P_{dyn|\ell} = \sum_{i=1}^{E_{max}} P_{dyn|\ell,i}\, q_i,$$

where $c_P(e)$ is the communication overheads associated with a task with execution time e under the proposed LS scheme, $n_\ell^e = \lfloor \frac{\ell}{c_P(e)} \rfloor$, and γ_j is obtained from Eq. (4.20) using $\alpha(T)$ in Eq. (4.21) and $\beta(T)$ in Eq. (4.23).

Figs. 4.6 and 4.7 are the plots of P_{dyn} vs. task arrival rate (λ), and $P_{dyn|\ell}$ vs. task laxity (ℓ), obtained from both the analytic models and simulation. Table 4.2 shows numerical examples of $P_{dyn|\ell}$ under different schemes. The conclusions inferred to from analytical results are similar to those from simulation results shown in Section 3.5.3, and hence are omitted here.

($\lambda = 0.8$) Task attributes	Lax. ℓ	No sharing	State probing	Random selection	Focused addressing	Proposed scheme	Quasi-perfect
$ET = \{0.4, 0.8,$	1	0.6184	0.1515	0.1286	8.649×10^{-2}	2.438×10^{-2}	5.247×10^{-3}
$1.2, 1.6\}_{0.25},$	2	0.4336	4.779×10^{-3}	2.302×10^{-3}	9.746×10^{-4}	3.034×10^{-4}	6.316×10^{-5}
$L = \{1, 2, 3\}_{1/3}$	3	0.2894	3.514×10^{-5}	1.447×10^{-5}	1.026×10^{-5}	7.156×10^{-6}	5.604×10^{-8}
$ET = \{0.027,$	1	0.7121	0.2476	0.1856	0.1524	4.342×10^{-2}	3.274×10^{-2}
$0.27, 2.703\}_{1/3},$	2	0.5896	5.086×10^{-2}	3.543×10^{-2}	2.249×10^{-2}	6.432×10^{-3}	2.316×10^{-3}
$L = \{1, 2, 3\}_{1/3}$	3	0.4923	4.994×10^{-3}	2.967×10^{-3}	9.594×10^{-4}	3.617×10^{-4}	1.604×10^{-4}
$ET = \{0.4, 0.8,$ $1.2, 1.6\}_{0.25},$ $L = \{1\}$	1	0.5894	0.1293	8.242×10^{-2}	7.153×10^{-2}	2.094×10^{-2}	5.946×10^{-3}

(a) $\lambda = 0.8$

($\lambda = 0.4$) Task attributes	Lax. ℓ	No sharing	State probing	Random selection	Focused addressing	Proposed scheme	Quasi-perfect
$ET = \{0.4, 0.8,$	1	0.1578	5.594×10^{-4}	7.645×10^{-4}	8.264×10^{-5}	1.187×10^{-5}	4.746×10^{-8}
$1.2, 1.6\}_{0.25},$	2	0.0479	6.402×10^{-6}	1.178×10^{-5}	9.536×10^{-7}	2.658×10^{-7}	1.042×10^{-10}
$L = \{1, 2, 3\}_{1/3}$	3	0.0176	2.782×10^{-7}	4.765×10^{-7}	4.846×10^{-8}	2.184×10^{-9}	0
$ET = \{0.027,$	1	0.2976	4.270×10^{-3}	5.247×10^{-3}	9.079×10^{-4}	2.035×10^{-4}	3.462×10^{-6}
$0.27, 2.703\}_{1/3},$	2	0.1846	2.346×10^{-5}	1.685×10^{-4}	9.896×10^{-6}	1.875×10^{-6}	1.395×10^{-8}
$L = \{1, 2, 3\}_{1/3}$	3	0.0796	2.693×10^{-7}	3.276×10^{-6}	7.903×10^{-8}	3.428×10^{-8}	0
$ET = \{0.4, 0.8,$ $1.2, 1.6\}_{0.25},$ $L = \{1\}$	1	0.1736	8.163×10^{-4}	5.943×10^{-4}	3.818×10^{-4}	6.547×10^{-5}	8.746×10^{-8}

(b) $\lambda = 0.4$

Table 4.2: $P_{dyn|\ell}$ vs. laxity ℓ for different task sets under different schemes ($N = 16$).

Arrival rate (λ)	Lax. ℓ	No sharing	State probing	Random selection	Focused addressing	Proposed scheme	Quasi-perfect
0.8	1	0.6184	3.027×10^{-2}	4.325×10^{-2}	2.878×10^{-2}	1.862×10^{-2}	5.247×10^{-3}
	2	0.4336	3.161×10^{-4}	2.946×10^{-4}	2.874×10^{-4}	2.389×10^{-4}	6.316×10^{-5}
	3	0.2894	2.763×10^{-6}	8.846×10^{-6}	5.323×10^{-7}	1.024×10^{-7}	5.604×10^{-8}
0.4	1	0.1578	1.875×10^{-7}	7.894×10^{-5}	4.275×10^{-7}	1.870×10^{-7}	4.746×10^{-8}
	2	0.0479	8.764×10^{-8}	3.376×10^{-6}	9.619×10^{-8}	3.678×10^{-9}	1.042×10^{-10}
	3	0.0176	4.763×10^{-10}	5.416×10^{-8}	5.136×10^{-10}	0	0

Table 4.3: $P_{dyn|\ell}$ for a task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$ under the ideal condition

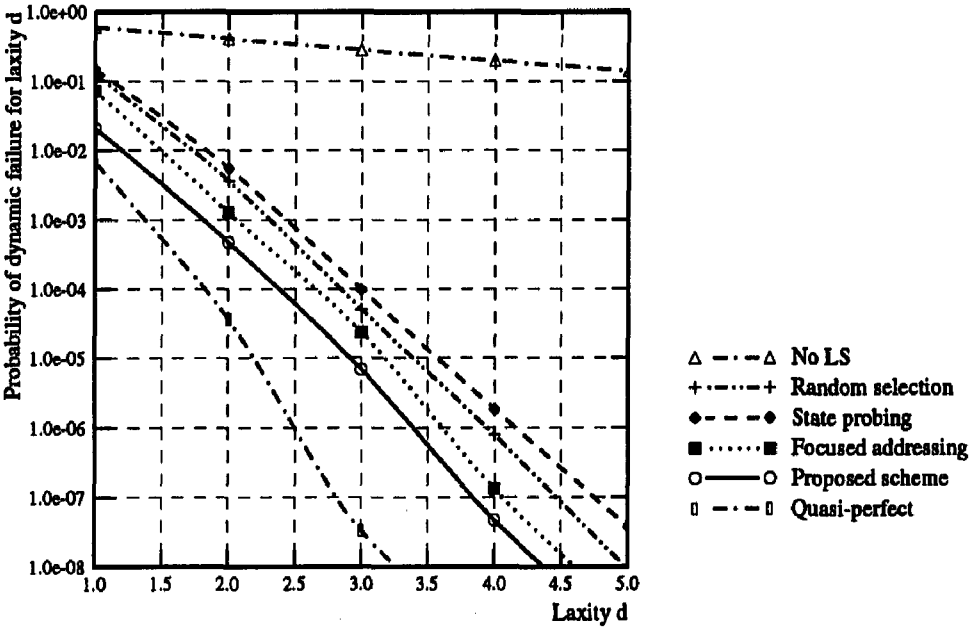


Figure 4.7: $P_{dyn|\ell}$ vs. task laxity ℓ for a 16-node system with a task set: $\lambda = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3, 4, 5\}_{0.2}$.

Mean Response Time

Probabilistically, the mean response time (MRT) is the sum of the average CET on a node, $\sum_{i=0}^{\infty} ip_T(i)$, and the average required execution time, $E(R) = 1$, i.e., $S = \sum_{i=0}^{\infty} ip_T(i) + 1$. It is conventionally used as a global system performance index in general-purpose distributed systems, and many approaches have been developed to minimize MRT. Table 4.4 gives MRT with respect to different task attributes under different schemes. Again, the conclusions drawn from analytical results are similar to those from simulation results in Section 3.5.3, and thus are omitted here.

Task Transfer-out Ratio

The task transfer ratio, r_{tt}, is defined as the fraction of arrived tasks (both external and transferred-in tasks) that have to be transferred out, and can be expressed as $r_{tt} = \sum_{j=1}^{L_{max}} \gamma_{j+1} q_j$. r_{tt} is a measure of the traffic overheads due to task transfers.

Table 4.5 gives r_{tt} for various task attributes under different schemes. Generally, r_{tt} increases as the system load gets heavier and/or the task laxity gets tighter. Moreover, as the variance of task execution time increases, the task transfer-out ratio increases. This is because a node easily becomes incapable of complete tasks in time upon the arrival of

Arrival rate (λ)	No LS	State prob.	Random selection	Focused address.	Prop. scheme	Quasi-perfect
0.2	1.154	1.117	1.115	1.117	1.118	1.108
0.4	1.406	1.302	1.268	1.265	1.257	1.236
0.6	1.868	1.498	1.483	1.466	1.446	1.439
0.8	3.521	1.872	1.836	1.789	1.720	1.668

(a) MRT vs. external task arrival rate for a task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$ under different LS schemes

($\lambda = 0.8$) Task attributes		No LS	State prob.	Random selection	Focused address.	Prop. scheme	Quasi-perfect
$L = \{1, 2, 3\}_{1/3}$	$ET = \{1\}$	3.024	1.796	1.782	1.763	1.704	1.687
	$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	3.521	1.872	1.836	1.789	1.720	1.668
	$ET = \{0.027, 0.27, 2.703\}_{1/3}$	6.106	2.174	2.101	1.979	1.822	1.804
$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	$L = \{1\}$	3.521	1.547	1.536	1.502	1.439	1.411
	$L = \{1, 2\}_{0.5}$	3.521	1.688	1.629	1.679	1.576	1.547
	$L = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	3.521	1.812	1.604	1.579	1.418	1.386

(b) MRT for different task sets under different LS schemes ($\lambda = 0.8$)

Table 4.4: Comparison of mean response time among different LS schemes.

($\lambda = 0.4$) Task attributes		No LS	State prob.	Random selection	Focused address.	Proposed scheme	Quasi-perfect
$L = \{1, 2, 3\}_{1/3}$	$ET = \{1\}$	1.348	1.267	1.264	1.259	1.248	1.240
	$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	1.406	1.302	1.268	1.265	1.257	1.236
	$ET = \{0.027, 0.27, 2.703\}_{1/3}$	1.806	1.350	1.335	1.314	1.316	1.301
$ET = \{0.4, 0.8,$ $1.2, 1.6\}_{0.25}$	$L = \{1\}$	1.406	1.198	1.184	1.165	1.167	1.154
	$L = \{1, 2\}_{0.5}$	1.406	1.250	1.242	1.233	1.228	1.214
	$L = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	1.406	1.211	1.190	1.162	1.163	1.148

(c) MRT for different task sets under different LS schemes ($\lambda = 0.4$)

Table 4.4: (continued) Comparison of mean response time among different LS schemes.

Arrival rate (λ)	State prob.	Random selection	Focused address.	Prop. scheme	Quasi-perfect
0.2	0.019	0.024	0.020	0.021	0.019
0.4	0.058	0.068	0.056	0.056	0.052
0.6	0.114	0.156	0.116	0.112	0.107
0.8	0.185	0.338	0.241	0.224	0.184

(a) r_{tt} vs. task arrival rate for the task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$ under different LS schemes

($\lambda = 0.8$) Task attributes		State prob.	Random selection	Focused address.	Prop. scheme	Quasi-perfect
$L = \{1, 2, 3\}_{1/3}$	$ET = \{1\}$	0.162	0.296	0.232	0.227	0.158
	$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	0.185	0.338	0.241	0.224	0.184
	$ET = \{0.027, 0.27, 2.703\}_{1/3}$	0.278	0.524	0.398	0.362	0.274
$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	$L = \{1\}$	0.306	0.463	0.355	0.348	0.286
	$L = \{1, 2\}_{0.5}$	0.221	0.383	0.286	0.266	0.216
	$L = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	0.289	0.454	0.367	0.322	0.282

(b) $\lambda = 0.8$

($\lambda = 0.4$) Task attributes		State prob.	Random selection	Focused address.	Prop. scheme	Quasi-perfect
$L = \{1, 2, 3\}_{1/3}$	$ET = \{1\}$	0.039	0.041	0.038	0.040	0.035
	$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	0.058	0.068	0.056	0.056	0.052
	$ET = \{0.027, 0.27, 2.703\}_{1/3}$	0.129	0.163	0.130	0.131	0.122
$ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	$L = \{1\}$	0.120	0.136	0.118	0.124	0.110
	$L = \{1, 2\}_{0.5}$	0.076	0.082	0.075	0.074	0.073
	$L = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$	0.121	0.154	0.123	0.131	0.120

(c) $\lambda = 0.4$

Table 4.5: Comparison of task transfer-out ratio among different LS schemes.

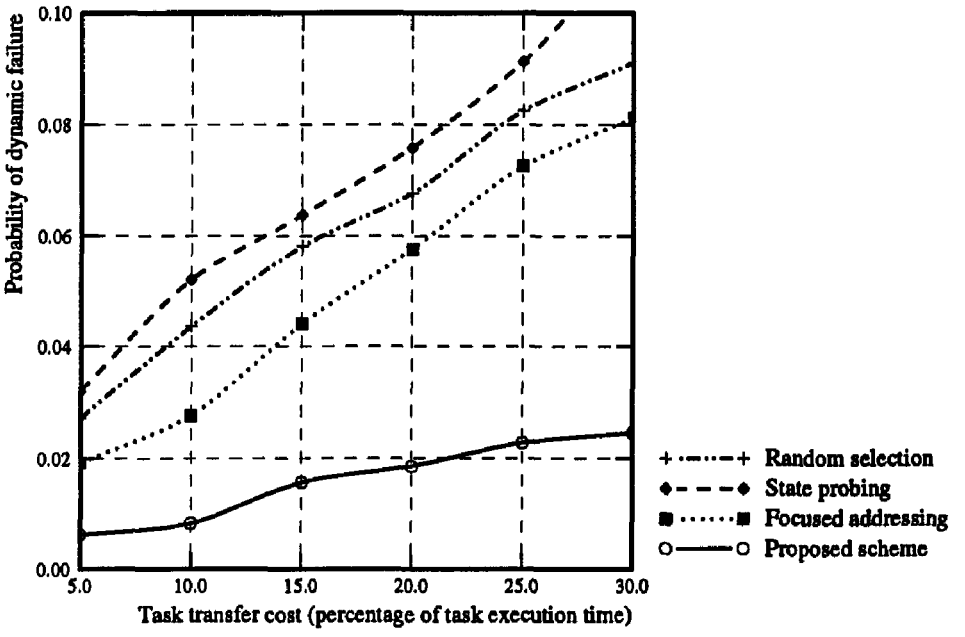


Figure 4.8: P_{dyn} vs. task transfer costs for a 16–node system with a task set: $\lambda = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{1/3}$.

long tasks or tasks with a *short* laxity, thus resulting in more task transfers under these conditions. Again, the observations made from analytical results are similar to those from simulation results in Section 3.5.3, and hence are omitted here.

Sensitivity to Communication Delays

As discussed in Section 3.5.3, there are two types of communication delay needed to be considered in LS: (i) the state–collection delay incurred from region–change broadcasts/state probes, where the queueing delay (or the queueing–related costs, o_3, o_4, and o_5) plays a dominating role; (ii) the delay associated with task transfers, where both the queueing delay and the transmission delay (or task transmission cost o_1) dominate. To study the impact of communication delay on the performance, P_{dyn} was computed for each scheme with (1) the task transmission costs being 5, 10, 15, and 20 % of the task execution time (i.e. $o_1 = 0.05, 0.10, 0.15, 0.20$), and (2) the queueing–related costs obtained from simulation (i.e. o_2, o_3, o_4, and o_5) being halved, doubled, and tripled. Figs. 4.8–4.9 (Table 4.6) show the effect of varying transmission delays and queueing delays on the performance of LS schemes. Similar conclusions can be inferred to from analytical results as well as from simulation results presented in Section 3.5.3.

($\lambda = 0.8$) Transfer costs	Lax. ℓ	State prob.	Random selection	Focused address.	Prop. scheme
5%	1	0.1090	8.074×10^{-2}	5.623×10^{-2}	1.845×10^{-2}
	2	3.257×10^{-3}	7.316×10^{-4}	3.924×10^{-4}	1.654×10^{-4}
	3	1.167×10^{-5}	5.239×10^{-6}	5.311×10^{-6}	1.987×10^{-6}
10%	1	0.1515	0.1286	8.649×10^{-2}	2.438×10^{-2}
	2	4.779×10^{-3}	2.302×10^{-3}	9.746×10^{-4}	3.034×10^{-4}
	3	3.514×10^{-5}	1.447×10^{-5}	1.026×10^{-5}	7.156×10^{-6}
15%	1	0.1834	0.1678	0.1328	3.725×10^{-2}
	2	7.524×10^{-3}	6.423×10^{-3}	2.678×10^{-3}	9.845×10^{-4}
	3	5.851×10^{-5}	3.675×10^{-5}	2.436×10^{-5}	1.436×10^{-5}
20%	1	0.2068	0.1892	0.1708	6.029×10^{-2}
	2	1.154×10^{-2}	7.467×10^{-3}	3.849×10^{-3}	1.346×10^{-3}
	3	1.878×10^{-4}	8.386×10^{-5}	5.014×10^{-5}	2.112×10^{-5}
25%	1	0.2550	0.2463	0.2212	6.486×10^{-2}
	2	1.394×10^{-2}	1.018×10^{-2}	8.746×10^{-3}	3.421×10^{-3}
	3	3.869×10^{-4}	1.846×10^{-4}	9.249×10^{-5}	6.857×10^{-5}

(a) Effect of task transfer costs on P_{dyn}

($\lambda = 0.8$) Delay coefficients	Lax. ℓ	State prob.	Random selection	Focused address.	Prop. scheme
halved	1	4.091×10^{-2}	8.415×10^{-2}	4.206^{-2}	1.978×10^{-2}
	2	3.758×10^{-4}	1.362×10^{-3}	6.270×10^{-4}	3.214×10^{-4}
	3	2.813×10^{-6}	7.145×10^{-6}	5.708×10^{-6}	3.758×10^{-6}
values from simulation	1	0.1515	0.1286	8.649×10^{-2}	2.438×10^{-2}
	2	4.779×10^{-3}	2.302×10^{-3}	9.746×10^{-4}	3.034×10^{-4}
	3	3.514×10^{-5}	1.447×10^{-5}	1.026×10^{-5}	7.156×10^{-6}
doubled	1	0.2134	0.1823	0.1538	3.129×10^{-2}
	2	2.801×10^{-2}	7.216×10^{-3}	1.537×10^{-3}	6.436×10^{-4}
	3	5.513×10^{-4}	2.875×10^{-5}	1.324×10^{-5}	9.578×10^{-6}
tripled	1	0.4194	0.2458	0.2173	4.245×10^{-2}
	2	7.475×10^{-2}	1.675×10^{-2}	1.267×10^{-2}	7.213×10^{-3}
	3	3.842×10^{-3}	2.334×10^{-4}	1.726×10^{-4}	2.046×10^{-5}

(b) Effect of queueing delays on P_{dyn}

Table 4.6: Effects of communication delay on P_{dyn} for a task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$ under different schemes.

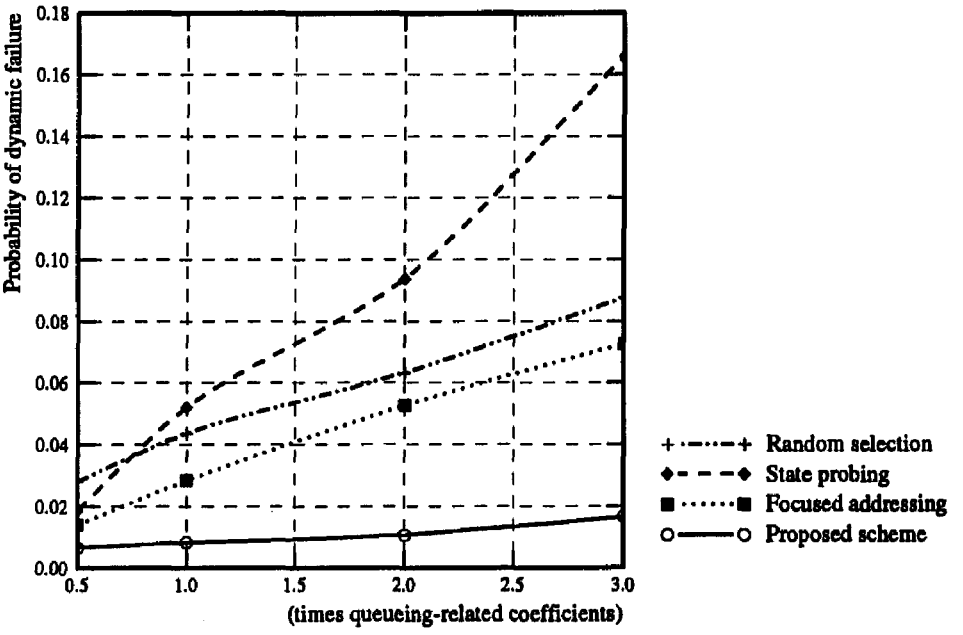


Figure 4.9: P_{dyn} vs. queueing delay coefficients for a 16-node system with a task set: $\lambda = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{1/3}$.

4.6 Conclusion

Queueing models are developed to quantitatively assess the proposed mechanism as well as three other schemes. Instead of the commonly-used QL, CET is used as the load state of each node. Each node's workload most relevant to a real-time task can thus be accurately modeled. Moreover, by including all computation/communication overheads, the proposed analytic models provide a means of evaluating the absolute real-time performance of LS schemes. The assumptions and approximations made in our analysis were checked with event-driven simulations.

Both the analytic and simulation results indicate that by using judicious exchange/use of state information and Bayesian decision mechanism, the proposed LS scheme though incurring more computation/communication overheads, makes a significant improvement in minimizing P_{dyn} over those simple LS schemes. This is in sharp contrast to the common notion that simple LS schemes yield performance close to that of complex ones for general-purpose systems where minimizing the mean response time is the main concern. Since missing a task deadline can cause a disastrous accident in a real-time environment, a more complex, but intelligent, LS scheme should be employed to minimize P_{dyn}.

We assumed a FCFS discipline on each node: a newly-arrived task is inserted at

Arrival rate per node	System attribute	Laxity ℓ	$P_{dyn\|\ell}$
$\bar{\lambda} = 0.8$	Homogeneous system ($\lambda_i = 0.8$, $1 \le i \le 64$)	1	3.326×10^{-3}
		2	8.798×10^{-6}
		3	1.266×10^{-7}
	Heterogeneous system ($\lambda_j = 0.65$, $\lambda_{16+j} = 0.95$, $\lambda_{32+j} = 0.65$, $\lambda_{48+j} = 0.95$, $1 \le j \le 16$)	1	1.470×10^{-3}
		2	5.855×10^{-6}
		3	9.872×10^{-8}
$\bar{\lambda} = 0.6$	Homogeneous system ($\lambda_i = 0.6$, $1 \le i \le 64$)	1	5.079×10^{-5}
		2	2.562×10^{-7}
		3	6.745×10^{-9}
	Heterogeneous system ($\lambda_j = 0.65$, $\lambda_{16+j} = 0.95$, $\lambda_{32+j} = 0.65$, $\lambda_{48+j} = 0.95$, $1 \le j \le 16$)	1	1.565×10^{-5}
		2	9.782×10^{-8}
		3	3.682×10^{-9}

Table 4.7: $P_{dyn|\ell}$ vs. task laxity ℓ for a task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ and $L = \{1, 2, 3\}_{1/3}$) in 64-node homogeneous/heterogeneous distributed systems.

the end of task queue if it can be completed in time on that node, and will otherwise be considered for transfer. To reduce P_{dyn}, the minimum–laxity–first–served (MLFS) discipline is shown to be better [HTT89] for queueing the incoming tasks at each node. That is, the tasks on a node are ordered by their laxities, and a task with the minimum laxity is always executed first by the node. If the MLFS discipline is employed, the transfer policy would not be simply of the threshold type. For example, a newly–arrived task may be inserted somewhere in the task queue, not necessarily at the end of the queue, thus possibly violating some of existing guarantees. (Such tasks, if possible, must be transferred to other capable nodes.) How to modify the parameters $\alpha(T)$ and $\beta(T)$ to account for this transfer policy is worth investigation.

Though we considered only homogeneous systems, the proposed scheme can also be applied to heterogeneous systems where different nodes may have different arrival rates of external tasks. Our simulation results indicate that the performance improvement is even more pronounced for heterogeneous systems than homogeneous ones (see Table 4.7). This is because that increasing the degree of heterogeneity increases the possibility that uneven task arrivals temporarily make some nodes incapable of completing tasks in time while leaving other nodes idle/underloaded. This situation can be effectively handled by using the proposed LS scheme. How to extend our analytic models to include the case for heterogeneous systems is an interesting, but difficult, matter.

CHAPTER 5

LS WITH CONSIDERATION OF FUTURE ARRIVALS

5.1 Introduction

Two issues must be considered in determining a candidate node for each overflow[1] task under the minimum–laxity–first–served (MLFS) discipline in a heterogeneous system:

G1: Minimization of the probability of transferring an overflow task $\mathcal{T}$ to an incapable node, i.e., the receiver of $\mathcal{T}$ should be one of those nodes which are observed/estimated to have sufficient resource surplus to guarantee $\mathcal{T}$.

G2: Avoidance of task collisions and/or excessive task transfers, and minimization of the possibility of a task's guarantee[2] being violated due to future tighter–laxity task arrivals.

We considered **G1** in Chapter 3 by using region–change broadcasts in the information policy, and preferred lists and Bayesian analysis in the location policy. **G2** need not be considered in homogeneous systems since the possibility of a task's guarantee being violated by future task arrivals is the same for all candidate receivers. The performance of LS can, however, be improved significantly by incorporating **G2** into LS decisions for heterogeneous systems. Consideration of **G2** is thus the main theme of this chapter.

The idea of not necessarily transferring a job to the node with the most resource surplus was first proposed by Yum and Schwartz [YS81, YL84] for routing messages in computer communication networks. Stankovic and Ramamritham [SRC85, RSZ89] considered the effect of future task arrivals on the guarantee of transferred–in tasks by (1) exchanging the information containing the percentage of free time among the nodes during

[1]Either a newly arrived task which cannot be completed in time on a node or the tasks queued on a node whose guarantees are deprived of by the insertion of a newly arrived tasks are termed as "overflow" tasks.

[2]As discussed in Chapter 1, by 'guarantee', we mean the node has enough resources to complete the task in time upon its arrival. A granted guarantee may be deprived later because of the arrivals of tighter–laxity tasks under the MLFS discipline.

the next *window*, the length of which is a tunable design parameter, and (2) using many parameters computed/estimated on-line to determine whether tasks will be transferred or not. They used heuristics and/or exhaustive search for on-line estimation/determination of parameters, and the effectiveness of their approach was evaluated via simulation of a small, six-node system without analytic modeling. By contrast, we shall take **G2** into account using a well-defined analytic framework, and the parameters needed for **G2** are updated on-line with Bayesian estimation theory.

In Chapter 3, we proposed a decentralized, dynamic LS mechanism which achieves **G1** in the presence of non-negligible communication delays by using the concept of buddy sets, region-change broadcasts [SC89a], and Bayesian decision theory. This LS mechanism will be used as an example of taking **G2** into account. Likewise, one can include **G2** in other existing LS schemes.

The rest of this chapter is organized as follows. Section 5.2 outlines the proposed LS mechanism, and formally defines **G1** and **G2** to be considered for choosing candidate nodes for real-time tasks to be transferred. Section 5.3 addresses the theoretical basis for **G2**. Section 5.4 discusses how the parameters needed for **G2** are estimated on-line using Bayesian estimation theory. Section 5.5 presents representative numerical examples. This chapter concludes with Section 5.6.

5.2 The Proposed Mechanism

In this section, we outline the proposed LS mechanism, and formally define the two issues, **G1** and **G2**, in choosing a candidate receiver for each overflow real-time task. The operations of a node's task scheduler which takes into account of future arrivals under the proposed mechanism are sketched in Fig. 5.1. To facilitate problem formulation and analysis, we define the following notation.

$\{p_i(j), j = 1, ..., E_{max}\}$: the distribution of composite[3] task execution time on node i, where E_{max} is the maximum task execution time. This distribution will be estimated on-line by each node i.

$\{\hat{p}_i(j), j = 1, ..., L_{max}\}$: the distribution of composite task laxity on node i, where L_{max} is the maximum laxity. This distribution will also be estimated on-line by each node i.

$CET_i(\ell)$: the cumulative execution time (CET) on node i contributed by tasks with laxity $\leq \ell$ under the MLFS discipline.

[3]both external and transferred

At each node n:
When a task T_i with execution time E_i and laxity ℓ_i arrives at node n:
determine the position, j_p, in the task queue $Q^{\dagger}$ such that $\ell_{j_p-1} \leq \ell_i \leq \ell_{j_p}$;
if current_time + $\sum_{k=1}^{j_p-1} E_k \geq \ell_i$ then
begin
receiver_node := table_lookup($\underline{Q}$:observation, ℓ_i:laxity)‡;
transfer task T_i to receiver_node;
change the recorded E_i to zero;
end
else
begin
queue task T_i at position j_p;
for $k = j_p + 1, length(Q)$
begin
if current_time + $\sum_{\ell=1}^{k-1} E_\ell \geq \ell_k$ then
begin
receiver_node := table_lookup($\underline{Q}$:observation, ℓ_k:laxity)‡;
dequeue and transfer T_k to receiver_node;
change E_k to 0, and modify $\{p_i(j)\}$; /* Section 5.4. */
end
end
if current_CET crosses TH_{2k}, $1 \leq k \leq \lceil \frac{K_i}{2} \rceil - 1$, then
/* region-change broadcasts: $TH_1, \cdots, TH_{K_i-1}$ are thresholds */
broadcast (1) time-stamped $CET_n(\ell)$'s, $\ell \in [0, L_{max}]$, and (2) λ_n, $\{p_n(j)\}$, and $\{\hat{p}_n(k)\}$ to all the other nodes in its buddy set;
end
$(\lambda_n, \{p_n(j)\}, \{\hat{p}_n(k)\})$ = parameter_update(E_i, ℓ_i, t_i:interarrival_time);

When a broadcast message arrives from node i, $1 \leq i \leq N$:
update observation of node i's state, $O_i(\ell), \ell \in [0, L_{max}]$;
record $(O_i(\ell), CET_i(\ell)), \ell \in [0, L_{max}]$ pair needed for
constructing probability distributions;
record λ_i, $\{p_i(j)\}$, and $\{\hat{p}_i(k)\}$;

At every clock tick:
current_CET := current_CET − 1;
if current_CET crosses TH_{2k}, $1 \leq k \leq \lceil \frac{K_i}{2} \rceil - 1$ then
broadcast (1) time-stamped $CET_n(\ell)$'s, $\ell \in [0, L_{max}]$, and (2) λ_n, $\{p_n(j)\}$, and $\{\hat{p}_n(k)\}$ to all the other nodes in its buddy set;

At every T_p clock ticks: /* probability and table update */
update probability distributions and the table of loss–minimizing decisions;

†The task queue Q is ordered by task laxities.
‡table_lookup is where the proposed location policy takes effect.

Figure 5.1: Operations of the task scheduler on each node.

$O_i(\ell)$: the observation about $CET_i(\ell)$ made by some node $j \neq i$.

$p_{C_i}(\cdot \mid O_i(\ell))$: the posterior distribution of $CET_i(\ell)$ given the observation $O_i(\ell)$. This posterior distribution is constructed by each node $j \neq i$ with the state samples collected via time–stamped region–change broadcasts, as discussed in Chapter 3.

$\overline{V}_{i,\ell}$: the event that future tighter–laxity task arrivals at node i do not invalidate the existing guarantee of a task with laxity ℓ.

$G_{i,\ell}$: the event that a task with laxity ℓ can be guaranteed (i.e., completed in time) by node i even in the presence of future tighter–laxity task arrivals.

The proposed LS mechanism which achieves both **G1** and **G2** works as follows: upon arrival of a task with laxity d at node n, the node uses the transfer policy described in Chapter 3 to check whether or not it can complete the task in time under the MLFS scheduling discipline, i.e., $CET_n(d) \leq d$. If it can, the task is accepted and queued at node n for execution. If the task cannot be completed in time locally or some of existing guarantees are to be violated by inserting the task into the node's schedule, the node looks up the list of best LS decisions and chooses — based on the current observation about other nodes' states, $\underline{O}$, and the laxity of the task(s) to be transferred — the best candidate receiver(s)[4] in its buddy set. The list of LS decisions is updated periodically based on both Bayesian analysis (described in Chapter 3) and queueing analyses.

Bayesian Analysis: is used to minimize the probability of transferring an overflow task $\mathcal{T}$ with laxity ℓ to an incapable node i given the observation at the time of locating the receiver of $\mathcal{T}$. The state information collected through state broadcasting/probing may become out–of–date due to the delays in collecting it. That is, a node's observation $O_i(\ell)$ may be different from $CET_i(\ell)$ at the time of making a LS decision. We countered this problem in Chapter 3 by using buddy sets, time–stamped region–change broadcasts, and Bayesian decision analysis. Succinctly, each node broadcasts a time–stamped message, informing all the nodes in its buddy set of a state–region change and all its on–line estimated parameters. Upon receiving a broadcast message from node i, each node in node i's buddy set updates its observation, and records the statistical samples which will be used to construct/update the posterior distribution, $p_{C_i}(\cdot \mid O_i(\ell))$, with Bayesian analysis. Each node estimates node

[4]If multiple tasks have to be transferred out (as a result of their guarantees being violated by the insertion of the newly arrived task), the observation about other nodes will be updated before making successive LS decisions. That is, if an overflow task with laxity ℓ and execution time m is to be transferred to node i, then node i's $CET_i(\ell)$ will be updated as $CET_i(\ell) + m$ before choosing candidate nodes for other tasks to be transferred.

i's true state based on its (perhaps out-of-date) observation via this posterior distribution of $CET_i(\ell)$ given $O_i(\ell)$. That is, each node — instead of hastily believing its observation about node i, $O_i(\ell)$ — estimates $CET_i(\ell)$ based on $O_i(\ell)$, and determines node i's LS capability via $p_{C_\ell}(\cdot \mid O_i(\ell))$. The sufficient condition for node i to be capable of completing a task in time with laxity ℓ is $CET_i(\ell) \leq \ell$, the probability of which can be calculated as:

$$P(CET_i(\ell) \leq \ell) = \sum_{k=0}^{\ell} p_{C_\ell}(k \mid O_i(\ell)).$$

Queueing Analysis: is used to minimize the probability of a task $\mathcal{T}$'s guarantee being deprived by subsequent tighter-laxity task arrivals during the period between the transfer (to node i) and the execution or the laxity of $\mathcal{T}$, whichever occurs first. We calculate this probability by:

$$P(\overline{V}_{i,\ell} \mid CET_i(\ell) \leq \ell) = \sum_{k=0}^{\ell} P(\overline{V}_{i,\ell} \mid CET_i(\ell) = k) \cdot p_{C_\ell}(k \mid O_i(\ell)),$$

where $p_{C_\ell}(k \mid O_i(\ell))$ is constructed in **G1**, and $P(\overline{V}_{i,\ell} \mid CET_i(\ell) = k)$ relates the effect of future tighter-laxity task arrivals to the guarantee of $\mathcal{T}$ with laxity ℓ. After $\mathcal{T}$ is transferred to node i, it has to wait for the execution of all the tasks which constitute $CET_i(\ell)$. Tag these tasks as 'primary' tasks. During the execution of primary tasks, 'secondary' tighter-laxity tasks may arrive, and have to be executed (or transferred out if they cannot be completed in time by node i) before $\mathcal{T}$. Similarly, there may be more tighter-laxity task arrivals during the execution of 'secondary' tighter-laxity tasks, and so on. Let X denote the total execution time contributed by the tighter-laxity tasks arrived at node i after the transfer of $\mathcal{T}$ but prior to the execution, or the laxity, of $\mathcal{T}$ whichever occurs first. $\mathcal{T}$ will be completed in time by node i in the presence of future task arrivals if $X \leq \ell - CET_i(\ell)$. We will derive $P(\overline{V}_{i,\ell} \mid CET_i(\ell) = k)$ in Section 5.3 using queueing analysis.

The parameters needed in calculating $P(\overline{V}_{i,\ell} \mid CET_i(\ell) \leq \ell)$ are the composite task arrival rate λ_i, the distribution of task execution time $\{p_i(.)\}$, and the distribution of task laxity $\{\hat{p}_i(.)\}$ on node i. Since the system state changes dynamically with time, these parameters have to be measured/estimated on-line by node i, and *piggybacked* in region-change broadcast messages to node n. Each node i records the interarrival time, the execution time, and the laxity of each task upon its arrival, and applies Bayesian estimation to determine the composite task arrival rate and the distributions of task execution time and laxity. Bayesian parameter estimation will be discussed in Section 5.4.

5.3 Consideration of Future Task Arrivals

We proposed in Chapter 3 a decentralized, dynamic LS mechanism that achieves **G1** by using the concept of buddy sets, time-stamped region-change broadcasts, and Bayesian decision analysis. The only refinement needed to consider **G2** is that now each time-stamped broadcast message contains two sets of information:

(1) Node number i, $CET_i(\ell)$, and the time t_0 when this message is sent.

(2) On-line estimated task characteristics: λ_i, $\{p_i(j), 0 \leq j \leq E_{max}\}$, and $\{\hat{p}_i(k), 1 \leq k \leq L_{max}\}$.

When the message broadcast by node i arrives at node n, node i's $CET_i(\ell)$ at t_0, can be recovered by node n. Node n can also trace back to find its observation about node i, $O_i(\ell)$, at time t_0. This observation $O_i(\ell)$ is what node n thought (observed) about node i when node i actually has $CET_i(\ell)$. $O_i(\ell)$'s along with $CET_i(\ell)$'s are used by node n to compute/update the posterior distribution, $p_{C_i}(\cdot \mid O_i(\ell))$, given the observation $O_i(\ell)$, once every T_p units of time. Besides, $CET_i(\ell)$ sent by node i at time t_0 is transformed into node n's new observation, $O_i(\ell)$, about node i at the time node n receives this message by the rule that $O_i(\ell) = k$ if $TH_k \leq CET_i(\ell) < TH_{k+1}$, $k \geq 0$, and $TH_0 \triangleq 0$. The second set of information is used to calculate the criterion of **G2**. Estimation of these parameters will be discussed in Section 5.4.

We now discuss how to incorporate **G2** in the proposed mechanism, and establish a theoretical basis for **G2**. Recall that $G_{i,\ell}$ denotes the event that an overflow task $\mathcal{T}$ with laxity ℓ is estimated to be completed in time by node i in the presence of future task arrivals, and $\overline{V}_{i,\ell}$, the event that future tighter-laxity task arrivals at node i will not invalidate the guarantee of $\mathcal{T}$. So,

$$\begin{aligned} P(G_{i,\ell} \mid O_i(\ell)) &= \sum_{k=0}^{\infty} [\, P(CET_i(\ell) \leq \ell) \cdot P(\overline{V}_{i,\ell} \mid CET_i(\ell) = k) \,] \cdot p_{C_i}(k \mid O_i(\ell)) \\ &= \sum_{k=0}^{\ell} P(\overline{V}_{i,\ell} \mid CET_i(\ell) = k) \cdot p_{C_i}(k \mid O_i(\ell)). \end{aligned} \tag{5.1}$$

The key issue here is how to derive $P(\overline{V}_{i,\ell} \mid CET_i(\ell) = k)$. Given $CET_i(\ell) \leq \ell$ at the time (say, time 0) of locating the receiver of an overflow task $\mathcal{T}$, the following two conditions may occur:

C1. Tighter-laxity tasks may arrive after the arrival of $\mathcal{T}$ at node i, and have to be executed before $\mathcal{T}$ under the MLFS discipline, thus increasing $CET_i(\ell)$.

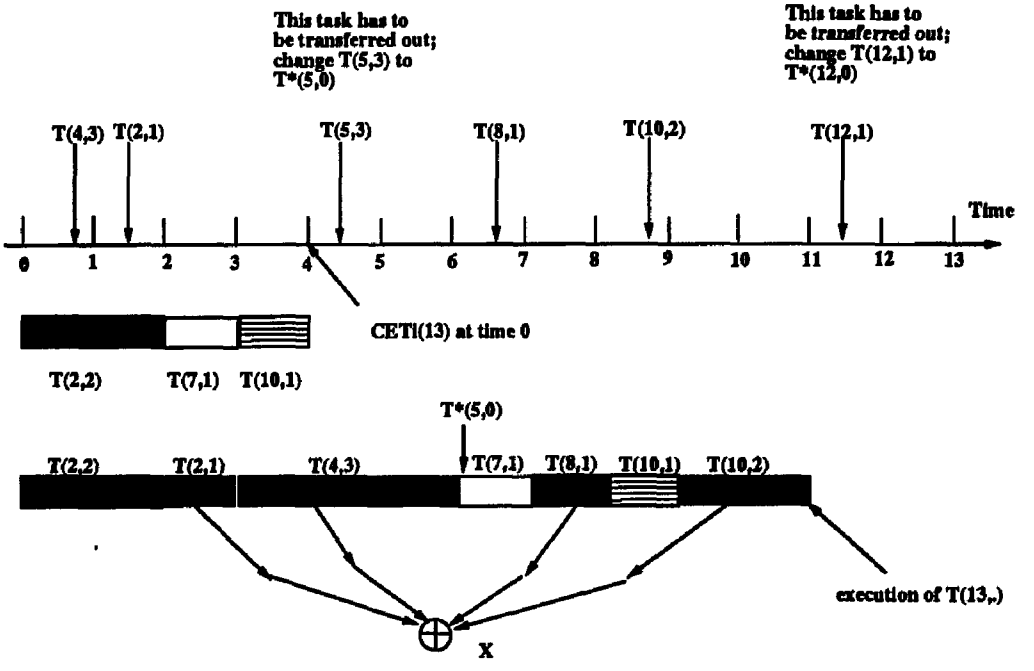
C2. The tasks constituting $CET_i(\ell)$ and/or some tighter–laxity tasks arrived later than $\mathcal{T}$ may get their existing guarantees violated due to the subsequent tighter–laxity task arrivals, and thus have to be transferred.

C2 violates the work conservation law commonly assumed in queueing analysis. To remedy this violation, we take into account the effect of **C2** on $CET_i(\ell)$ by adding $p_i(0)$ in the distribution of execution time, $\{p_i(j), 1 \leq j \leq E_{max}\}$. That is, when collecting statistics for the execution time distribution, each node i considers and records those tasks arrived at node i but eventually transferred out of node i as having null execution time, i.e., $p_i(0)$ is the fraction of tasks arrived at node i that will eventually be transferred out. Moreover, each future tighter–laxity task arrived at node i is estimated[5] to contribute j units of execution time with probability $p_i(j)$, $0 \leq j \leq E_{max}$, and must be executed before $\mathcal{T}$, where $j = 0$ represents the case when the guarantee of a task is deprived by subsequent tighter–laxity task arrivals. For example, in the first diagram of Fig. 5.2, the task $\mathcal{T}(5, 3)$ with execution time 3 and laxity 5 cannot be completed in time upon its arrival, and is thus treated by node i (in collecting its statistics) as a task $\mathcal{T}^*(5, 0)$ with execution time 0 and laxity 5.

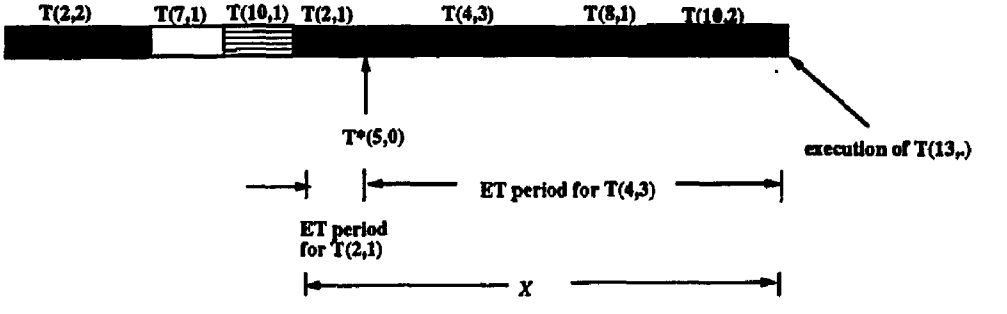
With this modification to $p_i(j)$'s, work conservation, which states that no tasks depart from node i before they are completely served, can be virtually retained in the subsequent derivation. Those tasks which have their guarantees violated because of tighter–laxity task arrivals are viewed as receiving zero unit of service time before they are transferred out of node i. On the other hand, because of this modification, our analysis does not model exactly the original queueing system of interest. However, since one cannot exactly predict the order of future task arrivals and their attributes and thus cannot know precisely whether or not a task will be transferred from the task queue, one has to resort to some statistical measure, e.g., $p_i(0)$, to take into account the effect of tasks being kicked out of the queue on calculating the distribution of X. Moreover, as our simulation results in Section 5.5.3 indicate, the performance of the proposed LS mechanism does significantly improve (by almost an order of magnitude) in reducing P_{dyn} with the approximate analysis. This is because the approximate distribution of X suffices to be an index of the likelihood of future tighter-laxity task arrivals at a node and its corresponding effect on the node's LS capability.

---

[5]Since we cannot really know the particular laxity and service requirements of future task arrivals in a dynamically changing distributed system with LS, we resort to the statistical measures, $p_i(j)$'s and $\hat{p}_i(j)$'s, based on the data gathered/estimated from the past to represent the attributes of future task arrivals on node i. $p_i(j)$'s reflect whether node i tends to receive long or short tasks; $\hat{p}_i(j)$'s reflect whether node i tends to receive tight or loose tasks.

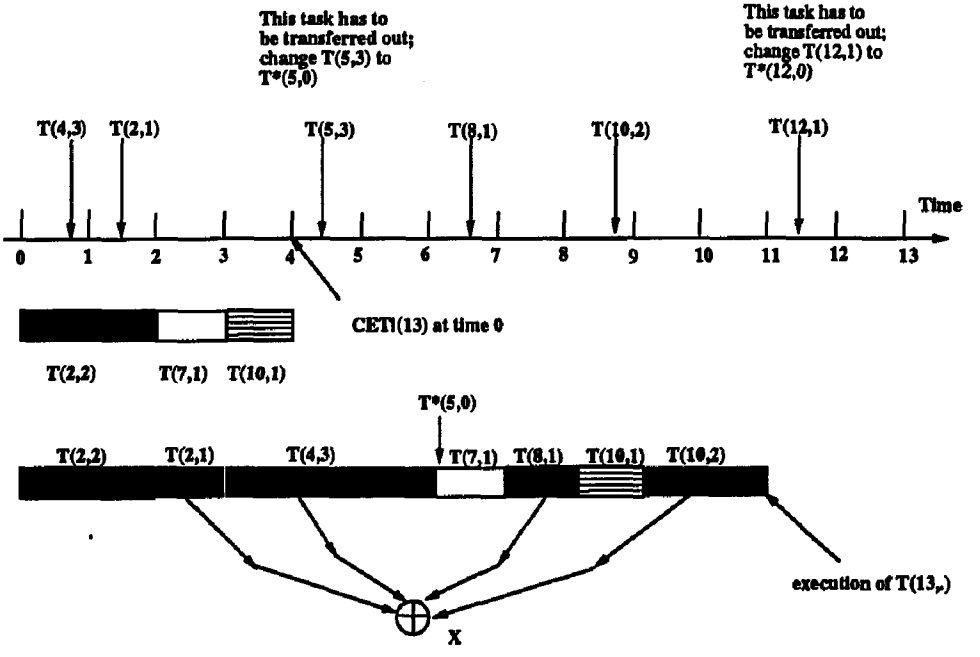


Actual order of service under MLFS discipline

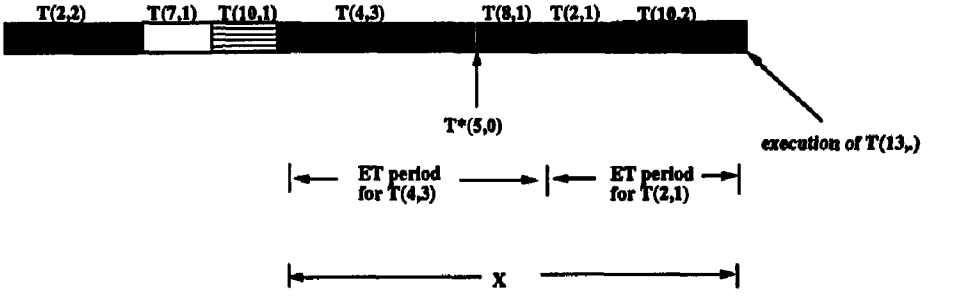


one order of service that may be used in the derivation of P(X <= x) (T(2,1) is executed first after CETl(13))

Figure 5.2: Future tighter–laxity task arrivals seen by a task T with $\ell = 13$. A task T with laxity x and execution time y is written as $T(x, y)$. This example shows (1) the independence of X from the execution order of tasks; (2) the definition and property of the ET period.

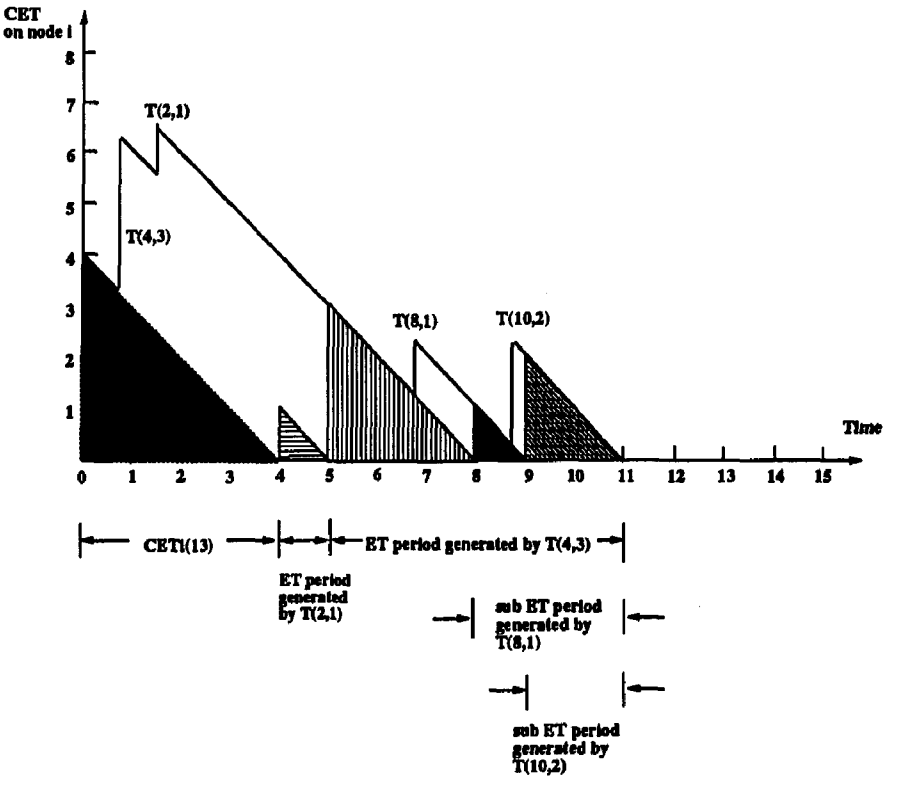


Actual order of service under MLFS discipline

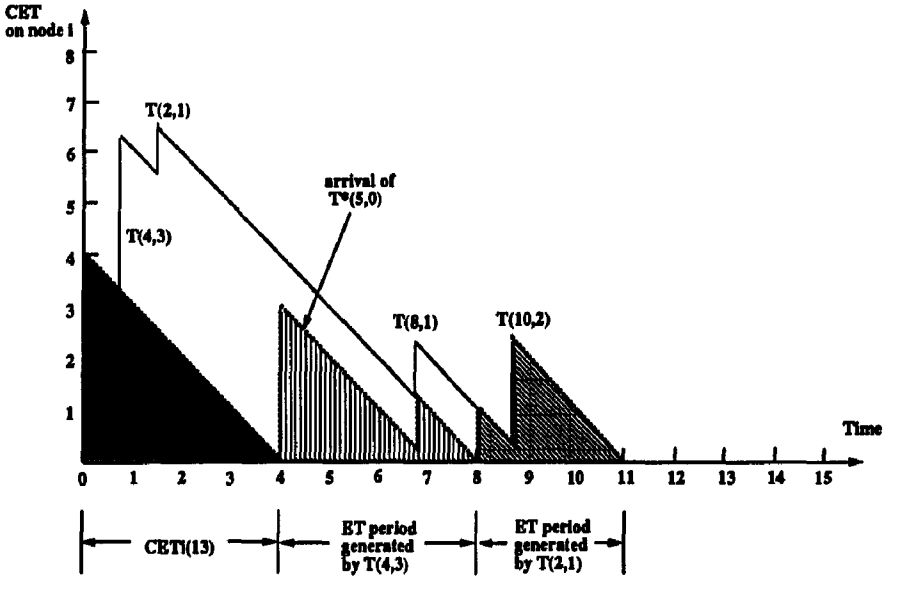


Another order of service that may be used in the derivation of P(X <= x)
(T(4,3) is executed first after CETI(13))

Figure 5.2: (continued) Future tighter–laxity task arrivals seen by a task $\mathcal{T}$ with $\ell = 13$.



(a) ET period for Fig. 5.2 (a)



(b) ET period for Fig. 5.2 (b)

Figure 5.3: ET periods for Fig. 5.2. Y-axis indicates the CET contributed by those tasks with laxity ≤ 13 on node i prior to the execution of task T.

Now, we want to derive $P(\overline{V}_{i,\ell} \mid CET_i(\ell) = k)$ subject to **C1**. Recall that X represents the total execution time contributed by tighter–laxity task arrivals at node i after time 0 or $\mathcal{T}$'s arrival but prior to $\mathcal{T}$'s execution (Fig. 5.2). If the distribution of X is known, then one can compute

$$P(\overline{V}_{i,\ell} \mid CET_i(\ell) = k) = P(X \leq \ell - k). \tag{5.2}$$

Since (1) node i stays busy (and cannot start execution of $\mathcal{T}$) as long as there are tasks with laxity $\leq \ell$ queued in front of it, and (2) the property of work conservation has been virtually retained with the way of collecting/estimating $p_i(j)$'s, e.g., no task will leave the system without getting serviced (but possibly with a null service time, the probability of which is estimated to be $p_i(0)$), we have virtually transformed the original system into a work-conserving one, and thus can use the well–known result of work–conserving systems [Kle75, GH85], which states the amount of work present in a work-conserving system does not depend on the service order of the customers. In our context, the amount of work seen by $\mathcal{T}$ prior to its execution, which is the sum of $CET_i(\ell)$ at time 0 (k in Eq. 5.2) and X, is independent of the service order of the tasks constituting k and X. Fig. 5.2 shows two examples of this independence. Note that in Fig. 5.2 a task with laxity x and execution time y is expressed as $T(x, y)$; the black blocks represent the tighter–laxity tasks arrived after the arrival of $\mathcal{T}$ but prior to the execution of $\mathcal{T}$; the other blocks represent the tighter–laxity tasks queued before the arrival of $\mathcal{T}$.

With the observation that the amount of work $k+X$ is independent of the execution order of tasks constituting k and X, we can permute the execution order of tasks with laxity $\leq \ell$ on node i so that those tasks contributing to $CET_i(\ell)$ may be executed first (Figs. 5.2 and 5.3). Then, we condition X on the number of tighter–laxity arrivals during $CET_i(\ell)$.

Let $\mathcal{S}^{<1>}$ denote the set of tighter–laxity tasks arrived during $CET_i(\ell)$. Each task $\mathcal{T}_m \in \mathcal{S}^{<1>}$ will contribute to X with tighter–laxity task arrivals during its execution. Denote the set of these arrivals as $\mathcal{S}^{<2>}_{\mathcal{T}_m}$. Furthermore, each task $\mathcal{T}_n \in \mathcal{S}^{<2>}_{\mathcal{T}_m}$ will also contribute to X with subsequent tighter–laxity task arrivals during its execution, which are represented by $\mathcal{S}^{<3>}_{\mathcal{T}_m,\mathcal{T}_n}$, and so on. This relation holds recursively for all m and n. We can thus view each tighter–laxity task arrived during $CET_i(\ell)$ as essentially generating its own execution time (ET) period. Examples of the ET period are shown in Figs. 5.2 and 5.3. Fig. 5.4 lists the corresponding $\mathcal{S}^{<i>}_{\mathcal{T}_{m_1},\mathcal{T}_{m_2},\ldots,\mathcal{T}_{m_{i-1}}}$. By the Markovian property[6] of the task arrival process, all ET periods have the same distribution. We characterize the ET period

[6]Validity of this approximation about composite task arrivals at each node will be discussed in Section 5.5.1 .

by its cumulative density function (CDF), $G(t)$, $t \geq 0$, whose expression can be derived using the above recursive relation and will be discussed in the next subsection. Then, we have

$$P(X \leq x \mid CET_i(\ell) = k) = \sum_{j=0}^{\infty} \{ \frac{(\lambda_h k)^j e^{-\lambda_h k}}{j!} \cdot G^{(j)}(x) \}, \tag{5.3}$$

where $G^{(j)}(x)$ is the CDF of the j–fold convolution of $G'(x) \triangleq \frac{dG(x)}{dx}$, and $\lambda_h = \sum_{n=1}^{\ell-1} \lambda \hat{p}_i(n)$ is the arrival rate of tasks with laxity $\leq \ell - 1$. The first factor inside the braces is the probability that j tasks with laxity $< \ell$ arrives within $CET_i(\ell) = k$, and the second factor $G^{(j)}(x)$ is the probability distribution of the sum of j ET periods.

Expression for $G(t)$

$G(t)$ is the CDF of the ET period generated by a tighter–laxity tasks arrived during $CET_i(\ell)$. By work conservation and the fact that a nonzero ET period continues to exist as long as there are unfinished tighter–laxity task arrivals after the beginning of the ET period (e.g., $S^{<i>}_{T_{m_1}, T_{m_2}, \ldots, T_{m_{i-1}}}$), $G(t)$ has the same distribution as the well–known busy period of an $M/G/1$ queue. (A *busy period* is defined to begin with the service (or equivalently, arrival) of a customer at an idle node and end when the node becomes idle again.) So, $G(t)$ can be readily expressed as [Ros83]:

$$G(t) = \sum_{n=1}^{\infty} \int_0^t e^{-\lambda_h t} \frac{(\lambda_h t)^{n-1}}{n!} b^{(n)}(t) dt, \tag{5.4}$$

where $b^{(n)}(t)$ is the n–fold convolution of $b(t)$, and $b(t)$ is the probability density function (PDF) of task execution time expressed as a continuous–time function, i.e.,

$$b(t) = \sum_{j=0}^{E_{max}} p_i(j) \cdot \delta(t - j);$$

here $\delta(.)$ is the impulse function.

Eq. (5.4) is an explicit expression of $G(t)$ in terms of known (or on–line estimated) quantities. The problem that arises in the infinite summation of both Eq. (5.4) and Eq. (5.3) can be solved by properly truncating high–order terms, and the error thus induced can be bounded by some predetermined value.

$\mathcal{S}^{<1>} = \{\mathcal{T}(4,3), \mathcal{T}(2,1)\}$,
$\mathcal{S}^{<2>}_{\mathcal{T}(2,1)} = \{\mathcal{T}^*(5,0)\}$, $\mathcal{S}^{<3>}_{\mathcal{T}(2,1),\mathcal{T}^*(5,0)} = \phi$,
$\mathcal{S}^{<2>}_{\mathcal{T}(4,3)} = \{\mathcal{T}(8,1)\}$, $\mathcal{S}^{<3>}_{\mathcal{T}(4,3),\mathcal{T}(8,1)} = \{\mathcal{T}(10,2)\}$,
$\mathcal{S}^{<4>}_{\mathcal{T}(4,3),\mathcal{T}(8,1),\mathcal{T}(10,2)} = \phi$.

(a) $\mathcal{S}^{<i>}_{\mathcal{T}_{m_1},\mathcal{T}_{m_2},\ldots,\mathcal{T}_{m_{i-1}}}$ corresponding to the order of service in Fig. 2(a).

$\mathcal{S}^{<1>} = \{\mathcal{T}(4,3), \mathcal{T}(2,1)\}$,
$\mathcal{S}^{<2>}_{\mathcal{T}(4,3)} = \{\mathcal{T}^*(5,0), \mathcal{T}(8,1)\}$, $\mathcal{S}^{<3>}_{\mathcal{T}(4,3),\mathcal{T}^*(5,0)} = \phi = \mathcal{S}^{<3>}_{\mathcal{T}(4,3),\mathcal{T}(8,1)}$,
$\mathcal{S}^{<2>}_{\mathcal{T}(2,1)} = \{\mathcal{T}(10,2)\}$, $\mathcal{S}^{<3>}_{\mathcal{T}(2,1),\mathcal{T}(10,2)} = \phi$.

(b) $\mathcal{S}^{<i>}_{\mathcal{T}_{m_1},\mathcal{T}_{m_2},\ldots,\mathcal{T}_{m_{i-1}}}$ corresponding to the order of service in Fig. 2(b).

Figure 5.4: $\mathcal{S}^{<i>}_{\mathcal{T}_{m_1},\mathcal{T}_{m_2},\ldots,\mathcal{T}_{m_{i-1}}}$ corresponding to the order of service in Fig. 3.

Figs. 5.5 and 5.6 give two numerical examples of $P(X \leq x \mid CET_i(\ell) = k)$, the former fixes x at 2 and the latter fixes k at 3. As expected, $P(X \leq x \mid CET_i(\ell) = k)$ decreases, for a fixed x, as the composite arrival rate of tighter–laxity tasks (λ_h) and/or the CET contributed by the queued tighter–laxity tasks (k) increases. Also, $P(X \leq x \mid CET_i(\ell) = k)$ increases with an increase x (or a decrease in λ_h) for a fixed k. Each node n can (1) use Eqs. (5.1)–(5.3) to compute the probability that an overflow task $\mathcal{T}$ with laxity ℓ is completed in time by node i with consideration of node i's future task arrivals, based on node n's observation about node i, $O_i(\ell)$, and (2) choose the node i with the largest $P(G_{i,\ell} \mid O_i(\ell))$ when transferring $\mathcal{T}$.

5.4 Parameter Estimation

One key issue in applying the proposed adaptive LS mechanism is the on–line estimation of the composite task arrival rate, λ_i, the distribution of composite task laxity, $\{\hat{p}_i(j)\}$, and the distribution of composite task execution time, $\{p_i(j)\}$. All on–line estimated parameters will then be conveyed to other nodes via region–change broadcasts. We discuss in this section how each node collects samples and makes on–line estimation of these parameters.

5.4.1 On–Line Estimation of Composite Task Arrival Rate

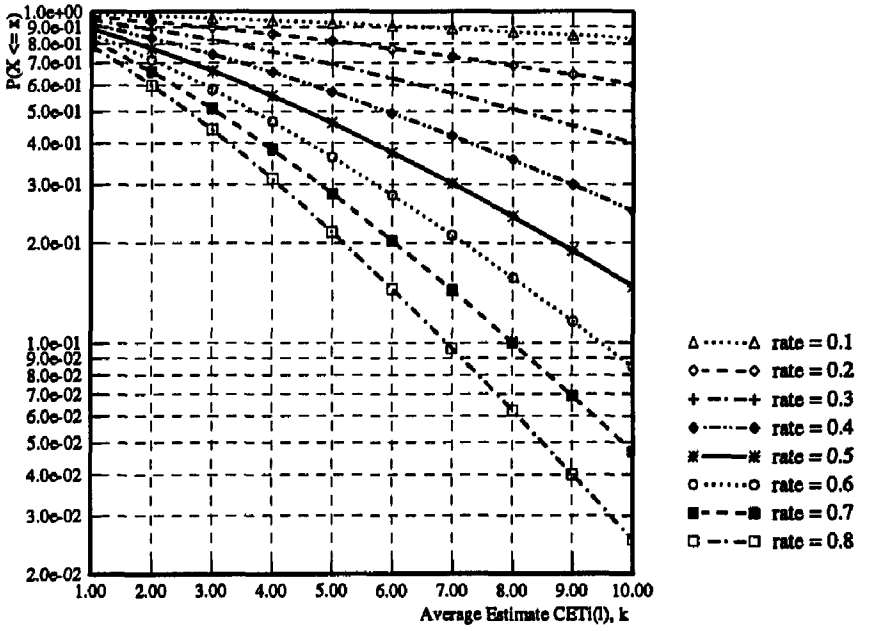


Figure 5.5: $P(X \leq 2 \mid CET_i(\ell) = k)$ for different values of k. $\lambda_h = 0.8\lambda$. Task execution time is exponentially distributed with 1.0.

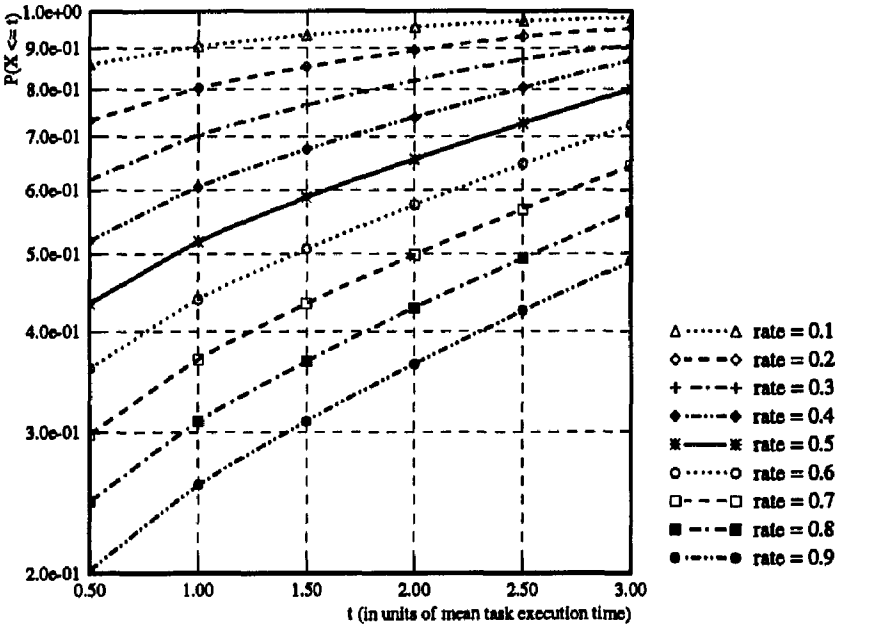


Figure 5.6: Conditional probability distribution of X given the estimated CET at node i is $k = 3$. $\lambda_h = 0.8\lambda$. Task execution time is exponentially distributed with 1.0.

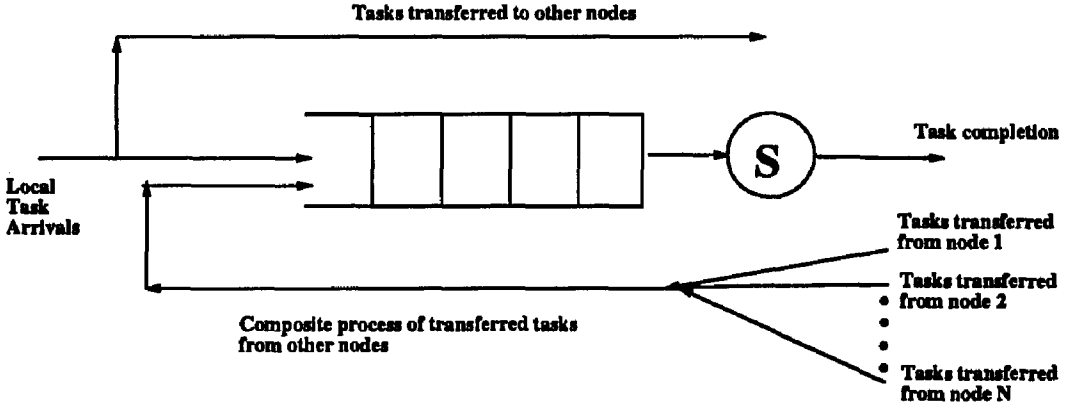


Figure 5.7: Task arrival and departure processes.

The composite task arrival process at a node is composed of the local (external) task arrivals and transferred–in task arrivals, the latter of which is itself a composite process of transferred–in tasks from different nodes (see Fig. 5.7). One difficulty in estimating the composite task arrival rate is that the transferred–in task arrival process (and thus the composite arrival process) may not be Poisson even if the local task arrival process is Poisson. This is because ($R1$) the probability of sending a task to (or receiving a task from) a node depends on the state of both nodes, making the splitting process non–Poisson, and ($R2$) task transmission times may not be exponentially distributed, making the process of transferred–in tasks non–Poisson. Furthermore, even if we assume the composite arrival process to exhibit behaviors similar to a Poisson process, the transferred–in task arrival rate from a node is not known due to the dynamic change of system state, which calls for the on–line estimation of the composite arrival rate.

Bayesian estimation is used for the on–line computation of the composite task arrival rate on a node. We consider the case of Poisson external task arrivals.[7] We further *approximate* the composite task arrival process to be Poisson (in spite of $R1$ and $R2$). This approximation rests on a general result of renewal theory which states that the superposition of increasingly many component processes (i.e., a reasonably large number of nodes) yields (in the limit) a Poisson process. We also ran simulations, collected task interarrival times on–line under the proposed LS mechanism, and used the Kolmogorov–Smirnov test to verify whether or not the Poisson approximation is valid. The simulation results show that for a

[7]We will later in the simulation consider the case of hyperexponential task interarrival times which represents a system potentially with bursty task arrivals, and investigate to what extent the proposed mechanism remains effective.

light to medium loaded system of size ≥ 12, this approximation holds. More on this will be discussed in Section 5.5.

Bayesian estimation works as follows [DeG70]: each node

1. monitors and records its task interarrival times continuously.
2. uses the noninformative distribution $g_1(\lambda_i) = const$, and $f(t \mid \lambda_i) = \lambda_i e^{-\lambda_i t}$ as its prior distribution and likelihood function, respectively.
3. computes the posterior distribution given the time sample t_k with
$$f(\lambda_i \mid t_k) = \frac{g_k(\lambda_i) \cdot f(t_k \mid \lambda_i)}{\int_0^\infty g_k(\lambda_i) \cdot f(t_k \mid \lambda_i) d\lambda_i}. \tag{5.5}$$
4. uses the posterior distribution $f(\lambda_i \mid t_k)$ for the current time sample t_k as the prior $g_{k+1}(\lambda_i)$ for the next time sample t_{k+1}.

To make the above method computationally manageable, it is desirable that both prior and posterior distributions belong to the same family of distributions. The major advantage of using a conjugate prior distribution in estimating λ_i (or any other parameters) is that if the prior distribution of λ_i belongs to this family, then for any sample size N_S and any values of the observed interarrival times, the posterior distribution of λ_i also belongs to the same family. Consequently, the calculation of Eq. (5.5) reduces essentially to updating the key parameters of a conjugate distribution. The interested readers are referred to [DeG70] for a detailed account of this.

For the composite arrival rate λ_i with an exponential sampling function, one can show that the γ–distribution

$$\mathcal{G}(\lambda \mid \alpha, \beta) = \begin{cases} \frac{\beta^\alpha}{\Gamma(\alpha)} \lambda^{\alpha-1} e^{-\beta\lambda}, & \text{for } \lambda > 0, \\ 0, & \text{otherwise.} \end{cases}$$

is its conjugate prior distribution, where $\Gamma(\alpha)$ is the gamma function such that $\Gamma(\alpha) = (\alpha - 1)!$ if α is integer. Specifically, given $\mathcal{G}(\lambda_i \mid \alpha = 1, \beta = t_1)$ as the prior γ–distribution, and given N_S interarrival time samples, $t_1, \ldots, t_{N_S}$, we have the posterior γ–distribution of λ_i as

$$\mathcal{G}(\lambda_i \mid \alpha = N_S, \beta = \sum_{i=1}^{N_S} t_i).$$

We use the mean of λ_i w.r.t. the posterior distribution as the estimated value which can be expressed in terms of the time samples only, i.e.,

$$E(\lambda_i) = \frac{N_S}{\sum_{k=1}^{N_S} t_k}. \tag{5.6}$$

Thus, the load information provided by the N_S latest interarrival–time samples can be easily abstracted by updating the key parameters in the conjugate distribution.

5.4.2 On–line Estimation of $p_i(j)$ and $\hat{p}_i(j)$

The other parameters needed for the proposed LS mechanism are $\{\hat{p}_i(j)\}$, and $\{p_i(j)\}$. The estimation techniques used to determine $\{\hat{p}_i(j)\}$ and $\{p_i(j)\}$ are virtually the same; we will henceforth concentrate on $\{\hat{p}_i(j)\}$.

We treat each task arrival as an experiment whose outcome belongs to one of L_{max} mutually exclusive and exhaustive categories, and $\hat{p}_i(j)$ as the probability that the outcome belongs to the j-th category ($1 \leq j \leq L_{max}$), where $\sum_{j=1}^{L_{max}} \hat{p}_i(j) = 1$. Suppose N_S independent experiment outcomes are available. Let $\boldsymbol{Y} = (Y_1, ..., Y_{L_{max}})$, where Y_j denotes the number of outcomes that belong to category j among these N_S outcomes. Then the likelihood function is a multinomial distribution with parameters N_S and $\boldsymbol{p} = (p_i(1), p_i(2), ..., p_i(L_{max}))$, i.e.,

$$f(\boldsymbol{y} \mid N_S, \boldsymbol{p}) = \begin{cases} \frac{N_S!}{y_1! \ldots y_{L_{max}}!} p_i(1)^{y_1} p_i(2)^{y_2} \ldots p_i(L_{max})^{y_{L_{max}}}, & \boldsymbol{y} \in \mathcal{N}^{L_{max}}, y_j \geq 0, \text{ for } 1 \leq j \leq L_{max}, \\ & \text{and } \sum_{j=1}^{L_{max}} y_j = N_S, \\ 0, & \text{otherwise.} \end{cases} \tag{5.7}$$

The conjugate family of distributions for the parameter $\boldsymbol{p}$ with a multinomial likelihood function is the Dirichlet distribution with parametric vector $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, ..., \alpha_{L_{max}})$, i.e.,

$$\mathcal{D}(\boldsymbol{p} \mid \boldsymbol{\alpha}) = \begin{cases} \frac{\Gamma(\alpha_0)}{\Gamma(\alpha_1) \ldots \Gamma(\alpha_{L_{max}})} p_i(1)^{\alpha_1 - 1} \ldots p_i(L_{max})^{\alpha_{L_{max}} - 1}, & \boldsymbol{p} \in \boldsymbol{R}^{L_{max}}, p_i(j) > 0, \text{ for } 1 \leq j \\ & \leq L_{max}, \text{ and } \sum_{j=1}^{L_{max}} p_i(j) = 1, \\ 0, & \text{otherwise,} \end{cases}$$

where $\alpha_0 = \sum_{i=1}^{L_{max}} \alpha_i$. Specifically, each node assumes the non-informative distribution as the prior distribution of $\boldsymbol{p}$, e.g., the prior distribution of $\boldsymbol{p}$ is the Dirichlet distribution with $\alpha_j = 1$, $1 \leq j \leq L_{max}$. After collecting N_S samples (i.e., after N_S task arrivals), and computing $(y_1, y_2, ..., y_{L_{max}})$, the posterior distribution of $\boldsymbol{p}$ is updated as

$$\mathcal{D}(\boldsymbol{p} \mid (\alpha_1 + y_1, \alpha_2 + y_2, ..., \alpha_{L_{max}} + y_{L_{max}})).$$

We then use the mean of $\boldsymbol{p}$ w.r.t. the posterior distribution as the estimated value, i.e., for $1 \leq j \leq L_{max}$,

$$E(p_i(j)) = \frac{\alpha_j + y_j}{\sum_{k=1}^{L_{max}} (\alpha_k + y_k)}.$$

Thus, the information provided by the most recent N_S task arrivals can be abstracted from the posterior distribution simply by updating the parameters.

5.5 Numerical Examples

The performance of the proposed LS mechanism is evaluated according to the following sequence:

- Validation of the Poisson approximation of the composite task arrival process which was made to facilitate the on–line estimation of task arrival rates.
- Elaboration of the parameters considered/varied in performance evaluation of LS schemes.
- Performance evaluation. We analyze the effect of considering **G2** on the performance of LS schemes. Second, we comparatively evaluate (1) no load sharing, (2) the focused addressing scheme [SRC85, RSZ89], (3) the proposed LS mechanism, and (4) quasi–perfect LS. Then, we study the impact of statistical fluctuation in task arrivals (by use of hyperexponential external task arrivals in the simulation) on the performance of the proposed mechanism.

5.5.1 On the Poisson Assumption of Composite Task Arrivals

The on–line estimation of the composite task arrival rate is done under the assumption that the composite task arrival process can be approximated to be Poisson.[8] This assumption is conjectured to become more realistic as the system size increases and/or as the system load gets lighter for the following reasons:

1. The superposition of increasingly many component processes yields (in the limit) a Poisson process. That is, as the system size gets larger, a node's state (CET) becomes less dependent on other nodes, the task transfer–out process at a node depends less on other nodes' states, and thus, the renewal assumption gets closer to reality.
2. In the case of Poisson external task arrivals, when the task transfer–out ratio is small, so is the "disturbance" to the (originally) Poisson arrival process caused by task transfers.

The validity of this approximation is checked by comparing the hypothesized exponential distribution and the sample cumulative distribution function. Given an estimate of the composite task arrivals being Poisson with arrival rate $\lambda = \frac{n}{\sum_{j=1}^{n} t_j}$, the Kolmogorov–Smirnov goodness–of–fit test is used to determine if $t_1, ..., t_n$ represent a random sample from an exponential distribution.

For completeness, we summarize below the steps of the Kolmogorov–Smirnov test used and discuss the data obtained from event–driven simulations. The interested readers are referred to [DeG86] for a detailed account of the Kolmogorov–Smirnov test. We first

[8] The same assumption was also used in [SC89a, MTS89b] without any justification.

run simulations and collect interarrival times on–line until $k = 100$ samples are obtained on each node. Second, we construct the sample (or empirical) distribution function $F_k(t)$ which is defined as the portion of the observed samples which are less than or equal to t, i.e., let $t_{(1)} < t_{(2)} ... < t_{(k)}$ be the values of the order statistics of the sample, then

$$F_k(t) = \begin{cases} 0, & t < t_{(1)}, \\ i/k, & t_{(i)} \leq t < t_{(i+1)}, \; i = 1, ..., k-1, \\ 1, & t = t_{(k)}. \end{cases} \tag{5.8}$$

Now we are interested in testing the following two hypotheses:

H_0: $t_1, t_2, ..., t_k$ is a random sample drawn from an exponential distribution with parameter λ, i.e., $F(t) \triangleq plim_{k \to \infty} F_k(t) = F_\lambda(t)$, where *plim* denotes "probabilistic limit";

H_1: H_0 is not true;

where $F_\lambda(t) = 1 - exp(-\lambda t)$ is the hypothesized exponential distribution. The test statistic D_{ks} for the Kolmogorov–Smirnov test is defined as the maximum difference between $F_k(t)$ and $F_\lambda(t)$, i.e.,

$$D_{ks} = \sup_{-\infty < t < \infty} | \, F_k(t) - F_\lambda(t)) \, | \, .$$

If D_{ks} is large, there are large differences between $F(t)$ and $F_\lambda(t)$, and the null hypothesis is rejected. To judge whether or not D_{ks} is large enough to justify rejecting H_0, we compare D_{ks} with the critical value D^*_{ks} [DeG86] of the Kolmogorov–Smirnov test. For example, as the sample size $k > 40$, D^*_{ks} can be calculated as $\frac{1.36}{\sqrt{k}}$ ($= 0.136$ in our case) at the significance level $\alpha_{ks} = 0.05$.[9] If $D_{ks} > D^*_{ks}$, we reject H_0; otherwise, we accept H_0 at the significance level α_{ks}.

It turns out that in the case of Poisson external task arrivals, we have $D < D^* = 0.136$ in the K–S test for all combinations of task attributes, when the number of nodes in the system ≥ 12, and/or the average task transfer–out ratio < 0.25 — this is always true in our simulations when the average external task arrival rate $\overline{\lambda^{ext}} = \frac{1}{K_{pn}} \sum_{i=1}^{K_{pn}} \lambda_i^{ext} \leq 0.8$, where K_{pn} is the number of processing nodes in the distributed system. Similarly, we have in the chi-square test, $\chi^2(obs) < \chi^2(0.05) = 7.81$ (i.e., H_0 is accepted) under the conditions specified above, where $\chi^2(obs)$, as D does in Kolmogorov–Smirnov test, measures the deviation of the empirical distribution from the hypothesized distribution, and $\chi^2(0.05)$ is the corresponding critical value at the significance level of 0.05. See Table 5.1 for numerical examples. Since both conditions are satisfied for the proposed LS mechanism,

[9] α_{ks} is the probability that H_0 is falsely rejected.

System size K_{pn}	Average system load $\overline{\lambda}$	Critical value D
8	0.2	0.084
	0.4	0.127
	0.6	0.187
	0.8	0.289
10	0.2	0.076
	0.4	0.092
	0.6	0.121
	0.8	0.203
12	0.2	0.063
	0.4	0.087
	0.6	0.104
	0.8	0.130
16	0.2	0.056
	0.4	0.081
	0.6	0.101
	0.8	0.117

Table 5.1: (a) Validation of the Poisson assumption with the Kolmogorov–Smirnov test: if $D < D^* = 0.136$, then the approximation is valid for the significance level 0.05.

the approximation of exponential interarrival times is acceptable at the significance level $\alpha_{ks} = 0.05$ for the case of Poisson external task arrivals.

5.5.2 Parameters Considered/Varied

The system configuration, the size of the buddy set, the tunable parameters chosen in the proposed LS mechanism, the computational overheads assumed, the ranges varied for task parameters (e.g., the average external task arrival rate per node, $\overline{\lambda_{ext}}$, the ratio of $\frac{e_{j+1}}{e_j}$, and the ratio of $\frac{t_{j+1}}{t_j}$), the confidence level achieved (in simulation), and the notation used all conform to those described in Section 3.5.

The transmission delay associated with each task transfer is varied from 10% to 50% of the execution time of each task being transferred. The broadcast–message–transmission delay is assumed to be negligible.[10] The medium–queueing delay which is experienced by both broadcast messages and transferred tasks and which dynamically changes with system load and traffic is modeled as a linear function of the number of tasks/messages queued for the particular medium.

The numerical experiments on the degree of system heterogeneity were conducted

[10]The physical transfer of the virtual memory image of a task may require tens of communication packets, while a region–change broadcast would in all likelihood need at most one packet.

System size K_{pn}	Average system load $\overline{\lambda}$	$\chi^2(obs)$
8	0.2	5.32
	0.4	6.49
	0.6	7.93
	0.8	8.06
10	0.2	4.68
	0.4	6.35
	0.6	7.42
	0.8	8.26
12	0.2	3.79
	0.4	4.23
	0.6	5.07
	0.8	6.12
16	0.2	2.86
	0.4	3.57
	0.6	4.35
	0.8	5.78

Table 5.1: (b) Validation of the Poisson assumption with the chi-square test: if $\chi^2(obs) = \sum_{j=1}^{C} \frac{(o_i - n_i)^2}{n_i} < \chi^2(0.05) = 7.81$, then the approximation is valid for the significance level 0.05. Note that n_i and o_i are obtained as follows. We first break up the domain of interarrival times (i.e., $(0, \infty)$) into $C = 5$ categories. Under the assumption that $\lambda e^{-\lambda t}$ governs interarrival times, we determine the number, n_i, of t_i's that are expected to fall into category i. Second, we count the number, o_i, of the k=100 time samples obtained from the simulation which actually fall into category i.

by dividing nodes into K_n groups; the nodes in each group g, $1 \leq g \leq K_n$, have an external task arrival rate λ_g^{ext} such that $\frac{\lambda_{g+1}^{ext}}{\lambda_g^{ext}} = r_n$ and $\frac{1}{K_{pn}} \sum_{i=1}^{K_{pn}} \lambda_i^{ext} = \overline{\lambda_{ext}}$, where K_{pn} is the number of processing nodes in the distributed system. The performance of the proposed LS mechanism was simulated while varying K_n from 2 to 6, and r_n from 2 to 4.

The case with hyperexponential interarrival times represents a system potentially with bursty task arrivals, and is used to study the impact of statistical fluctuation in task arrivals on the LS performance. The squared coefficient of variation of hyperexponential arrivals (CV^2) is varied from 1 to 64.

5.5.3 Performance Evaluation

Instead of using the mean task response time as the performance metric, we use the probability of dynamic failure, P_{dyn}, maximum system utilization, $\overline{\lambda_{ext}}$, (i.e., the highest frequency of task activations allowed for a specified P_{dyn}), the task transfer–out ratio, r_{tt}, and the frequency of task collision, f_{tc}. Moreover, we present only those results that we

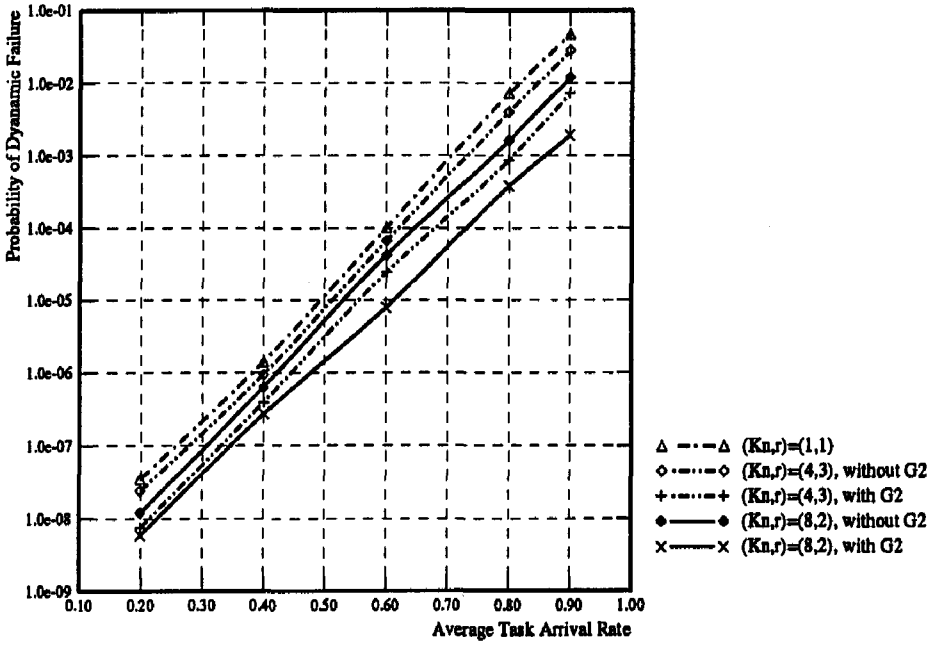


Figure 5.8: P_{dyn} of the proposed LS approach with and without **G2** for a task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and $L = \{1, 2, 3\}_{1/3}$. The task transfer delay is assumed to be 10% of task execution time.

believe are the most relevant, interesting, and/or representative.

Effects of G2 on the Performance of LS Algorithms: We now analyze the performance improvement achievable by considering **G2**, and compare the performance of the proposed LS policy with others using trace–driven simulations.

Fig. 5.8 plots the performance of the proposed LS policy *with* and *without* consideration of **G2** for different degrees of system heterogeneity. When $\overline{\lambda_{ext}} \geq 0.4$ and as the degree of system heterogeneity increases, one can make a substantial performance gain with **G2**. In other words, inclusion of **G2** in a LS scheme avoids the possibility of transferring tasks to those nodes which tend to become overloaded or receive tighter–laxity tasks. This, in turn, reduces the possibility of task collisions and task re-transfers (and thus P_{dyn}). See Table 5.2 for numerical examples of f_{tc} for the proposed mechanism.

Taking **G2** into account is not restricted to the proposed LS mechanism; it can also be incorporated into other existing LS schemes. For example, a parallel state–probing approach can be modified to reduce P_{dyn} as follows. Each node collects and estimates λ_i, $\{p_i(j), 0 \leq j \leq E_{max}\}$, and $\{\hat{p}_i(k), 1 \leq k \leq L_{max}\}$ on–line as discussed in Section 5.4.

λ_{ext}	$(K_n, r_n) = (2,2)$		$(K_n, r_n) = (4,3)$		$(K_n, r_n) = (8,2)$	
	with G2	without G2	with G2	without G2	with G2	without G2
0.2	0.018	0.021	0.019	0.027	0.021	0.034
0.4	0.047	0.056	0.050	0.072	0.062	0.087
0.6	0.074	0.118	0.078	0.132	0.097	0.146
0.8	0.176	0.224	0.189	0.247	0.201	0.269

Table 5.2: f_{tc} of the proposed approach with and without **G2** for a task set $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and $L = \{1,2,3\}_{1/3}$. The task transfer delay is assumed to be 10% of task execution time.

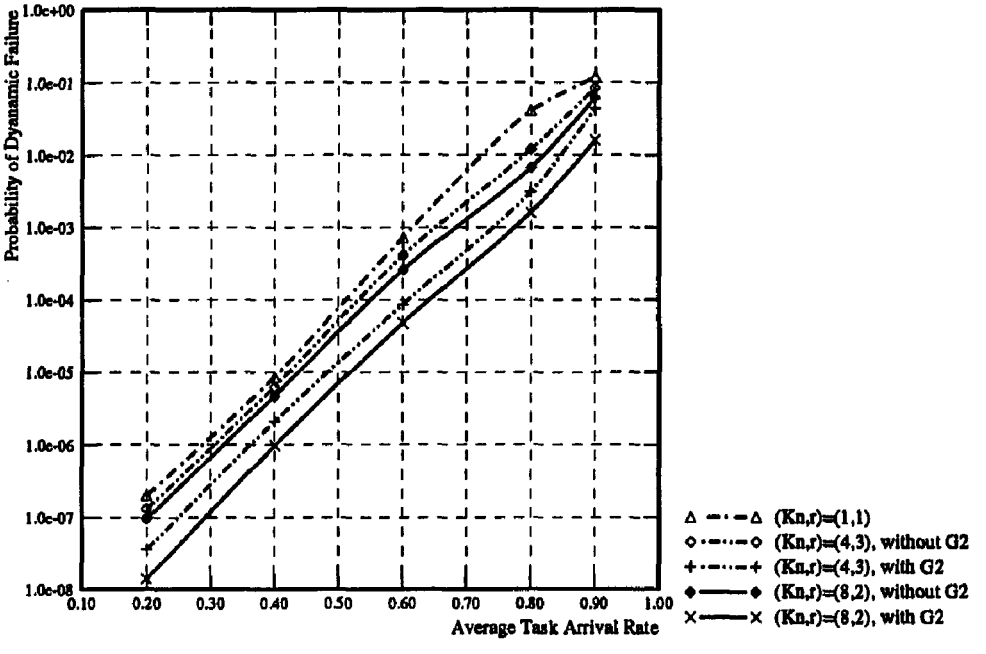


Figure 5.9: P_{dyn} of the parallel state probing with and without **G2** for a task set with $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, and $L = \{1,2,3\}_{1/3}$. The task transfer delay is assumed to be 10% of task execution time.

A node with an overflow task probes a predetermined number[11] of nodes in parallel. A probed node i sends the probing node, in addition to its CET, the estimated λ_i, $p_i(j)$'s, and $\hat{p}_i(k)$'s. After receiving this information, the probing node considers **G2** and chooses the most capable receiver. Fig. 5.9 depicts the performance of the parallel state–probing approach with and without consideration of **G2**. Again, the performance improvement made by **G2** becomes substantial as the degree of system heterogeneity increases.

Performance Comparison among Different LS Algorithms: The proposed LS mechanism is comparatively evaluated against a simplified version of the focused addressing approach in [SRC85, RSZ89]. We also compare the proposed LS mechanism with two baseline schemes, i.e., no LS and quasi–perfect LS. Figs. 5.10–5.12 show the performance curves of different LS schemes for different task attributes. Three task sets are considered: (I) $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{1/3}$; (II) $ET = \{0.027, 0.27, 2.7\}_{1/3}$, $L = \{1, 2, 3\}_{1/3}$; and (III) $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1\}_1$. The average external task arrival rate $\overline{\lambda_{ext}}$ is varied from 0.2 to 0.9. Fig. 5.13 shows the effect of task–transfer delay on the performance of different LS mechanism. For clarity, only the performance curves corresponding to $(K_n, r_n) = (1, 1)$ are shown for no LS and perfect LS.

From these curves, one can observe that the proposed LS mechanism outperforms the focused addressing approach in minimizing P_{dyn}, especially when (1) the distribution of external task laxity is tight, (2) the spectrum of task execution time is wide, (3) the degree of system heterogeneity is large, and (4) the task–transfer delay is significant. The superiority of the proposed LS policy under condition (1) comes from the fact that an overflow task with tight laxity cannot tolerate the possibility of being transferred to an incapable node or a node which will become incapable in near future. (Note that the proposed LS mechanism deliberately eliminates such a possibility.) Under condition (2), a node easily becomes incapable with the arrival of even a single task which has a tight laxity and requires a large execution time. This makes the consideration of future task arrivals crucial in locating the receiver of each overflow task. The performance improvement under (3) and (4) results from the consideration of **G2** and **G1**, respectively.

Fig. 5.14 shows the plot of maximum system utilization $\overline{\lambda_{ext}}$ versus ϵ. This relates the worse–case achievable P_{dyn} to the frequency of task activations. One important result from these curves is that with the clever use/interpretation of state information/statistical samples, we do not have to sacrifice $\overline{\lambda_{ext}}$ for lower P_{dyn}, which is in contrast to the common

[11] This has been set to 5 in our simulations based on the finding in [ELZ86].

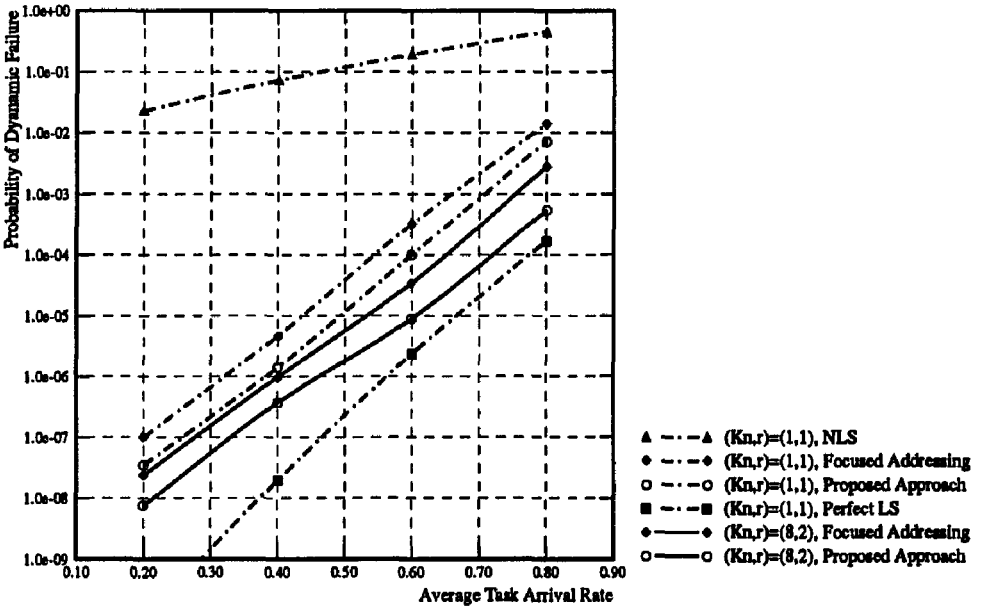


Figure 5.10: Performance comparison w.r.t. P_{dyn} among different LS approaches for a 16-node system with a task set $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{1/3}$. The task transfer delay is assumed to be 10% of task execution time.

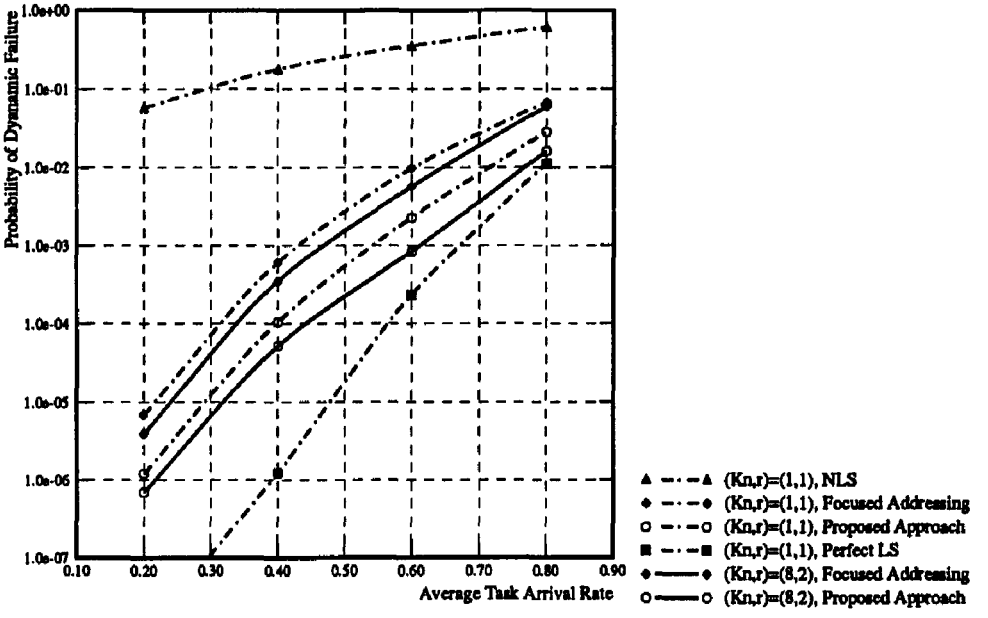


Figure 5.11: Performance comparison (w.r.t. P_{dyn}) among different LS approaches for a 16-node system with a task set $ET = \{0.027, 0.27, 2.7\}_{1/3}$, $L = \{1, 2, 3\}_{1/3}$. The task transfer delay is assumed to be 10% of task execution time.

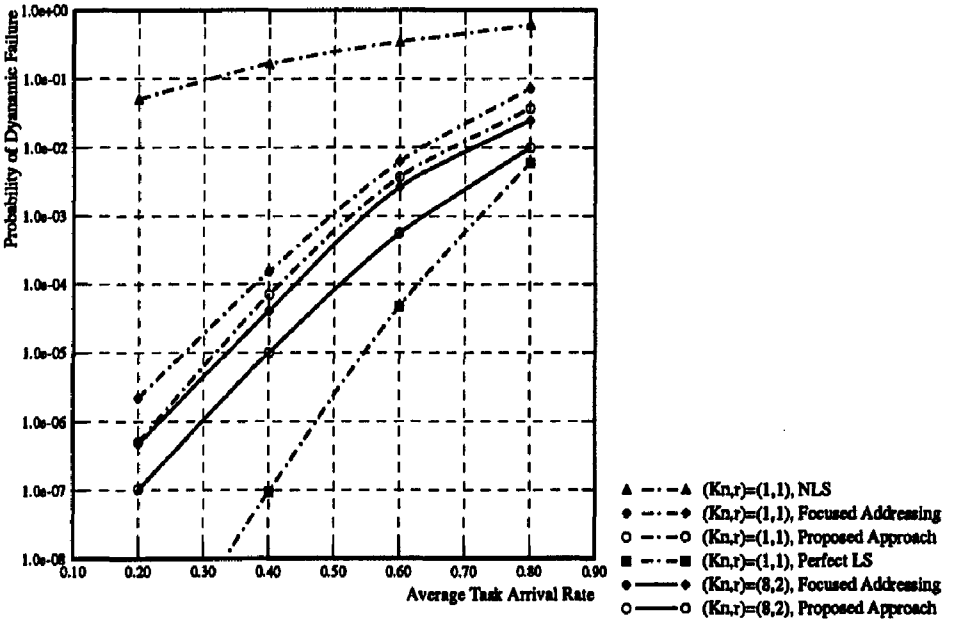


Figure 5.12: Performance comparison (w.r.t. P_{dyn}) of different LS approaches for a 16–node system with a task set $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1\}_1$. The task transfer delay is assumed to be 10% of task execution time.

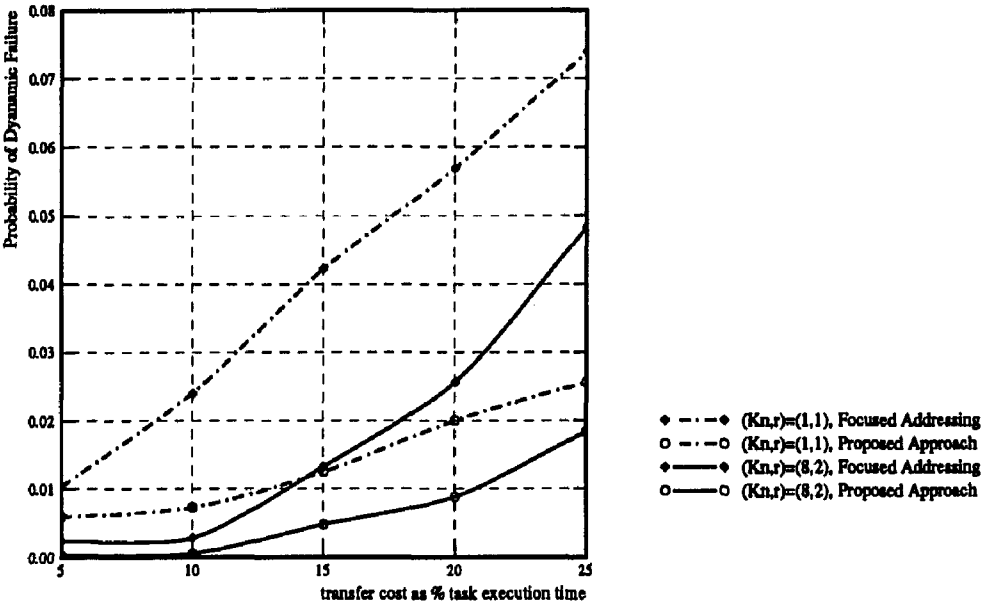


Figure 5.13: Effect of task transfer delay on P_{dyn} for the proposed approach and the focused addressing approach in a 16–node system with $\overline{\lambda_{ext}} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{1/3}$.

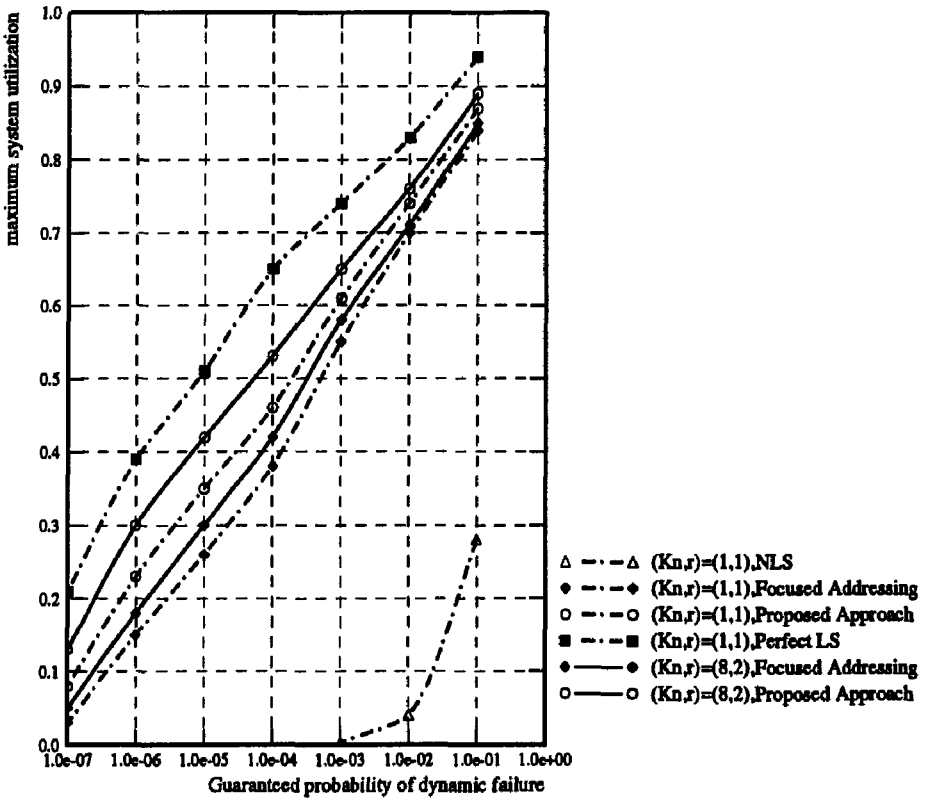


Figure 5.14: $\overline{\lambda_{ext}}$ vs. ϵ. External task arrivals are Poisson. Both task execution time and laxity are exponentially distributed.

notion of trading system utilization for real–time performance. Moreover, the proposed mechanism outperforms the focused addressing scheme, and the performance superiority becomes more visible as the degree of system heterogeneity increases.

Table 5.3 gives numerical results of task transfer–out ratio r_{tt} for different LS schemes. From this table, we observe that r_{tt} for the proposed LS approach is smaller than that for the focused addressing. This indicates the ability of the proposed LS mechanism in avoiding task transfers to (i) incapable nodes as a result of using out–of–date state information, and (ii) capable nodes that may easily become incapable as a result of future task arrivals. The performance improvement becomes more pronounced as the tightness of task laxity distribution and/or the degree of system heterogeneity increases.

Effect of Statistical Fluctuation in Task Arrivals on the Proposed Scheme: One issue in using the Bayesian estimation technique is to what extent the proposed LS mechanism remains effective when the attributes of tasks arrived at a node randomly fluctuate. We study this effect on the estimation of composite task arrival rates by simulating task sets

LS mechanism	Focused Addressing		Proposed Algorithm		Perfect LS	
(K_n, r_n)	(1,1)	(8,2)	(1,1)	(8,2)	(1,1)	(8,2)
Task set I	0.241	0.278	0.206	0.237	0.184	0.213
Task set II	0.398	0.416	0.304	0.311	0.276	0.294
Task set III	0.355	0.392	0.321	0.347	0.286	0.312

Table 5.3: Performance comparison (w.r.t. task transfer–out ratio) of different LS approaches for a 16–node system. $\overline{\lambda_{ext}} = 0.8$. The task transfer delay is assumed to be 10% of task execution time.

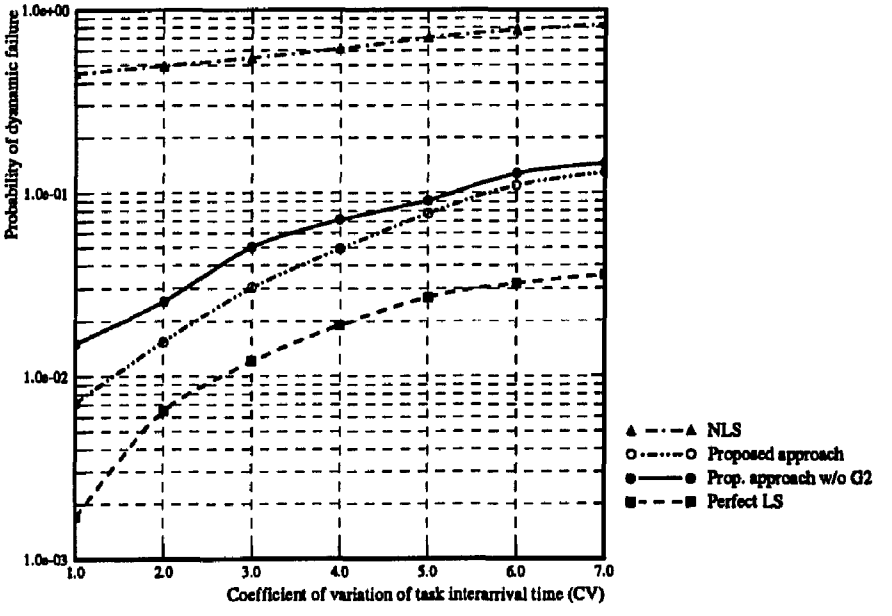


Figure 5.15: P_{dyn} vs. coefficient of variation (CV) of external task interarrival times for a 16–node homogeneous ($(K_n, r_n) = (1,1)$) system with $\overline{\lambda_{ext}} = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1,2,3\}_{1/3}$.

with different hyperexponential task interarrival times (e.g., by varying the coefficient of variation (CV) of hyperexponential task interarrival times). Fig. 5.15 shows the simulation results under heavy system load ($\overline{\lambda_{ext}} = 0.8$, where the LS performance is most sensitive to the variation of CV) with the window of the sample size $N_S = 30$. From Fig. 5.15, we draw the following conclusions: as the variance of task interarrival times (CV) becomes greater, the sample–mean based estimate gets worse. This is because the variability effect due to task burstiness cannot be totally smoothed out. This accounts, in part, for the performance degradation of the proposed mechanism. Another reason for performance degradation is due to the capacity limit of the distributed system; that is, the system inherently cannot complete simultaneously bursty time–constrained tasks in time. The proposed LS mechanism remains effective (despite its gradual degradation) up to CV=7.42 (or $CV^2 = 55$) beyond which it reduces essentially to the scheme without the use of Bayesian estimation. This suggests that within a wide range of statistical fluctuation in task arrival patterns, parameter estimates based on the Bayesian technique suffice to serve as an index of the tendency of future task arrivals on a node.

5.6 Conclusion

We enhanced the LS mechanism proposed in Chapter 3 by considering the effects of future task arrivals on locating the best receiver for each overflow task. The proposed LS mechanism minimizes not only the probability of transferring an overflow task $\mathcal{T}$ to an incapable node, but also the probability of the chosen remote node failing to complete $\mathcal{T}$ in time because of the node's future arrivals of tighter–laxity tasks. Consideration of future task arrivals significantly improves the performance of LS (in minimizing P_{dyn}) when system workloads are unevenly distributed among nodes.

All parameters needed in the LS decision process — observation/estimation of other nodes' states, composite task arrival rates of other nodes, and task execution time and task laxity distributions of other nodes — are collected/estimated on–line using time–stamped region–change broadcasts and Bayesian estimation theory. This makes the proposed mechanism (1) less sensitive to communication delays and (2) adaptive to dynamically varying workloads with little computational overhead.

The Poisson approximation of composite task arrivals — which has been used without justification in other LS schemes (e.g., [SC89a, MTS89b]) — has been carefully checked by the Kolmogorov–Smirnov goodness–of–fit test. Our simulation results have indicated that this assumption holds for a system with a reasonably large (≥ 12) number of

nodes and/or with a small (≤ 0.25) average task transfer–out ratio. The negative impact of statistical fluctuation in task arrivals on the proposed approach with use of Bayesian estimation is also shown to be tolerable within a wide range of bursty task arrivals (e.g., up to $CV^2 = 55$, where CV is the coefficient of variation of external task interarrival times used in the simulation).

CHAPTER 6

INCORPORATION OF OPTIMAL TIMEOUTS INTO LS

6.1 Introduction

As discussed in Chapter 3, for the LS scheme that uses state–region change broadcasts as its information policy, each node i broadcasts a message, informing the other nodes in its buddy set of a stage–region change *whenever* its CET crosses a certain broadcast threshold [SH91]. To detect node failure and to prevent sending overflow tasks to failed nodes, a timeout mechanism is usually incorporated into this kind of LS scheme as follows. Each node n makes the transfer and location decisions as specified by the LS scheme. In addition, node n considers node i failed if it has not heard from node i for the timeout period, $T_{out}^{<i>}$, since its receipt of node i's latest broadcast, and will henceforth not send its overflow task(s) to node i even if node i is observed (through the state information gathered in region–change broadcasts) to be capable of completing the task(s) in time. Obviously, the determination of $T_{out}^{<i>}$ is crucial to the performance of the timeout mechanism, and is the main subject of this chapter.

There are two possible scenarios of node n not receiving any region–change broadcast from node i for $T_{out}^{<i>}$:

S1. Node i failed sometime after issuing its last broadcast message;

S2. Task arrival and completion/transfer activities alternate in such a way that the state or CET of node i oscillates within two adjacent broadcast thresholds, or remains in a broadcast-threshold interval.

The occurrence of **S1** is determined by the failure rate of node i, while **S2** is determined by task arrival, completion, or transfer activities on node i, all of which dynamically change with the composite task arrival rate, the attributes of tasks arrived at node i, and node i's initial state node. Some simple techniques could be used to determine whether **S1** or **S2** occurs: node n may determine whether node i failed or not by probing it at the time of

making a LS decision, but in such a case, it has to wait for node i's response before making the LS decision. This could introduce unacceptable delays to those tasks to be transferred, the negative effect of which increases significantly with communication delay [MTS89b].

On the other hand, node n may arbitrarily choose a fixed timeout period *a priori*. In this case, node n runs the risk of (1) hastily and falsely diagnosing a healthy node as failed if the timeout period chosen is too small and (2) failing to detect node failures in a timely manner if the chosen period is too large. Actually, as will be demonstrated in Section 6.5.2, the best value of $T_{out}^{<i>}$ varies drastically with the system load, the attributes of tasks arrived at node i, and the state node i was initially in, since task arrival and completion/transfer activities on a node dynamically vary with these parameters. (We will compare the performance of using the best $T_{out}^{<i>}$ against that of using some pre-specified timeout period in Section 6.5.3.) This calls for a timeout mechanism which on–line collects/estimates these parameters and dynamically adjusts $T_{out}^{<i>}$ accordingly. That is, the timeout mechanism requires each node i to collect statistics, estimate on–line its "composite"[1] task arrival rate and distributions of task execution time and laxity, and convey the estimated parameters to other nodes in its buddy set by piggy-backing them in state–region change broadcasts. This information will then be used by the other nodes to calculate $T_{out}^{<i>}$.

One key issue in designing a timeout mechanism with on–line adjustable timeout periods is to express $T_{out}^{<i>}$ as a function of task attributes and load states. Since the determination of $T_{out}^{<i>}$ involves a tradeoff between the performance improvement gained by reducing $T_{out}^{<i>}$ (thus enabling early detection of a node failure) and the performance degradation resulting from hasty, incorrect diagnoses, we formulate this problem as a hypothesis testing (HT) problem with two hypotheses, and determine $T_{out}^{<i>}$ by maximizing the probability of detecting node failures subject to a pre-specified probability of incorrect diagnosis.

To further reduce the probability of incorrect diagnosis, the timeout mechanism is modified as follows: each node n calculates the "best" timeout period, $T_{out}^{<n>}$, for itself (as well as for other nodes), and broadcasts its state not only at the time of state–region changes but also when it has remained within a broadcast-threshold interval and has thus been silent for $T_{out}^{<n>}$. That is, with a few extra, timely broadcasts, the undesirable effect of incorrect diagnosis can be reduced while enabling fast detection of node failures.

The LS mechanism in Chapter 3 will be used here as an example to demonstrate how to incorporate the proposed timeout mechanism into a LS scheme with aperiodic state–

[1] both external and transferred-in

change broadcasts. One can, of course, include the timeout mechanism in other existing LS schemes.

The rest of the chapter is organized as follows. Section 6.2 outlines the proposed timeout mechanism. Section 6.3 and 6.4 establishes a theoretical basis for the calculation of optimal $T_{out}^{<i>}$. The HT formulation is treated in Section 6.3, while the probability distribution needed in the HT formulation is derived in Section 6.4 by applying the randomization technique to a continuous–time Markov chain which characterizes the state evolution. Section 6.5 presents and discusses representative numerical examples, and this chapter concludes with Section 6.6.

6.2 The Proposed Mechanism

We proposed in Chapter 3 a decentralized, dynamic LS mechanism for distributed real–time systems without considering node failures. In this section, we state the assumptions made about the system and the analytical derivation to be performed, and then discuss how to incorporate the proposed timeout mechanism in distributed LS to tolerate node failures.

We assume that the node clocks in the system are synchronized to establish a global time-base. A scheme for achieving this synchronization was presented in [RKS90]. We also assume that the underlying communication subsystem supports reliable broadcasts [RS88, KS91b] so that a non-faulty node can correctly broadcast its state change to all other non-faulty nodes in the system. Finally, each node is assumed to have a constant exponential failure rate λ_F. (This assumption is commonly used in reliability evaluation [EB86, AAS86].)

To facilitate mechanism description and analysis, we introduce the following notation and assumptions:

λ_i: the composite (external and transferred–in) task arrival rate at node i. We *approximate* the composite task arrival process to be Poisson, the validity of which has been treated in Section 5.5.1. This approximation is used to facilitate the derivation of $T_{out}^{<i>}$ and the on-line estimation of parameters needed for calculating $T_{out}^{<i>}$.

$\{p_i(j), 1 \leq j \leq E_{max}\}$: the distribution of execution times of composite tasks at node i, where E_{max} is the maximum task execution time. This distribution will be estimated on–line by each node i.

$\{\hat{p}_i(j), 1 \leq j \leq L_{max}\}$: the distribution of laxities of composite tasks at node i, where L_{max} is the maximum laxity. This distribution will also be estimated on-line by each node

i.

CET_i: the cumulative task execution time (CET) on node i.

$\underline{T_Q} = (T_1; T_2; ...; T_{L_{max}})$: the record for task execution times of the sorted queue on a node, where $T_j \triangleq e_1^j e_2^j ... e_{j+1}^j$ is an execution-time record of tasks with laxity $j \in \{1, \ldots, L_{max}\}$ currently queued on a node,[2] and $e_k^j \in \{0, \ldots, E_{max}\}$, $1 \leq k \leq j+1$, is the execution time required by the k-th task among those laxity–j tasks in the queue. ($e_k^j = 0$ if there are less than k laxity–j tasks in the queue.)

$T_{out}^{<i>}$: the timeout period; node i will be diagnosed as failed if no broadcast message from node i has been received for this period since the receipt of its latest broadcast.

The operations of a node's task scheduler which employs the LS mechanism described in Chapter 3 and the timeout mechanism are given in Fig. 6.1. The timeout mechanism to be incorporated into LS is composed of the following sub-mechanisms.

On–line Parameter Estimation: node i records on-line the inter-arrival time, the required execution time, and the laxity of each task upon its arrival, and applies the Bayesian technique to estimate the task parameters: λ_i, $\{p_i(j), 1 \leq j \leq E_{max}\}$, and $\{\hat{p}_i(j), 1 \leq j \leq L_{max}\}$. Application of the Bayesian technique to estimate these parameters has been treated in Section 5.4. These estimated parameters are piggy-backed with the description of the sorted task queue $\underline{T_Q}$ in region–change broadcasts.

Determination of Timeout Periods and Detection of Node Failures: upon receiving a message broadcast by node i, node n uses the task parameters and $\underline{T_Q}$ contained in the message to calculate $T_{out}^{<i>}$. A theoretical basis for determining $T_{out}^{<i>}$ will be established in Sections 6.3 and 6.4 by using the hypothesis testing (HT) and randomization techniques. Conceptually, the problem of determining $T_{out}^{<i>}$ is first formulated as a HT problem by making a tradeoff between **S1** and **S2**. Then, the key expression needed in the HT formulation, i.e., the probability distribution that no message has been received from node i within time t given that node i is operational is derived by first modeling the state evolution of node i as a continuous–time Markov chain and then applying the randomization technique on the constructed Markov chain to derive the distribution of interest.

Node n considers node i failed if it has not heard from node i (via region–change broadcasts) for $T_{out}^{<i>}$ since node i's latest broadcast, and will not transfer any overflow tasks

[2]The reason that T_j is of the form $e_1^j e_2^j ... e_{j+1}^j$ is because a node can queue, under the MLFS discipline, at most $j+1$ tasks with laxity j, in which case all but the last laxity–j task require 1 unit of execution time, and there are no tighter–laxity tasks queued at the node.

At each node n:
When a task T_i with execution time E_i and laxity ℓ_i arrives at node n:
    determine the position, j_p, in the task queue Q such that $\ell_{j_p-1} \leq \ell_i \leq \ell_{j_p}$;
    if current_time + $\sum_{k=1}^{j_p-1} E_k \geq \ell_i$ then
     begin
        receiver_node := table_lookup($\underline{Q}$:observation, ℓ_i:laxity);
        transfer task T_i to receiver_node;
     end
    else
     begin
        queue task T_i at position j_p;
        for $k = j_p + 1, length(Q)$
          begin
            if current_time + $\sum_{\ell=1}^{k-1} E_\ell \geq \ell_k$ then
             begin
                receiver_node := table_lookup($\underline{Q}$:observation, ℓ_k:laxity);
                dequeue and transfer T_k to receiver_node;
             end
          end
        if current_CET crosses TH_{2k}, $1 \leq k \leq \lceil \frac{K_t}{2} \rceil - 1$, then
          begin /* region-change broadcasts */
            broadcast (1) time-stamped CET_n's, and (2) λ_n, $\{p_n(j)\}$, $\{\hat{p}_n(k)\}$
             to all the other nodes in its buddy set;
            calculate $T_{out}^{<n>}$ and reset timeout_clock$_n$;
          end
     end
    $(\lambda_n, \{p_n(j)\}, \{\hat{p}_n(k)\})$ = parameter_update(E_i, ℓ_i, t_i:interarrival_time);
When a broadcast message arrives from node i:
    update observation of node i's state, O_i;
    if node i is disabled then
      enable node i;
    else
      record (O_i, CET_i) pair needed for Bayesian decision analysis;
    calculate $T_{out}^{<i>}$ using λ_i, $\{p_i(j)\}$ and $\{\hat{p}_i(k)\}$, and reset timeout_clock$_i$;
At every clock tick:
    current_CET := current_CET $-$ 1;
    if (current_CET crosses TH_{2k}, $1 \leq k \leq \lceil \frac{K_t}{2} \rceil - 1$) or (timeout_clock$_n$ expires)
    then
      begin
       broadcast (1) time-stamped CET_n's, and (2) λ_n, $\{p_n(j)\}$, and $\{\hat{p}_n(k)\}$
        to all the other nodes in its buddy set;
       calculate $T_{out}^{<n>}$ and reset timeout_clock$_n$;
      end
    if timeout_clock$_i$ expires then
      disable node i;
At every T_p clock ticks: /* table update */
    update the table of loss-minimizing decisions by Bayesian decision analysis;

Figure 6.1: Operations of the task scheduler on each node.

to node i until it receives a broadcast message from node i again. Whenever a failed node i is recovered, it broadcasts its recovery to all the other nodes in its buddy set. Upon receiving such a broadcast message, node n will consider node i capable of receiving tasks if the subsequent region–change broadcasts indicate so. On the other hand, node n also calculates its own timeout period $T_{out}^{<n>}$ at the time of broadcasting a state-region change. If node n has remained within a broadcast-threshold interval and has been silent for $T_{out}^{<n>}$, it broadcasts an extra message to inform other nodes of its fault-free (or 'I am alive') status.

6.3 Determination of the Optimal Timeout Period

In this section and the next section, we will establish a theoretical basis for the determination of $T_{out}^{<i>}$. The problem of determining $T_{out}^{<i>}$ is formulated as a HT problem in this section. The probability distribution needed to solve the HT problem is then derived using the randomization technique in Section 6.4. For a node n to determine the best timeout period of node i, $T_{out}^{<i>}$, it needs two sets of parameters, both of which are contained in the most–recently–received broadcast message from node i:

1. on–line estimation of λ_i, $\{\hat{p}_i(j), 1 \leq j \leq L_{max}\}$, and $\{p_i(j), 1 \leq j \leq E_{max}\}$.
2. node i's sorted task queue, $\underline{T_Q}$.

We discussed how a node estimates the parameters of task attributes (i.e., λ_i, $\hat{p}_i(j)$'s, and $p_i(j)$'s) in Section 5.4. $\underline{T_Q}$ is the record of the task execution times of the sorted queue at node i. How a node broadcasts its on–line estimated parameters and $\underline{T_Q}$ to all the other nodes in its buddy set in Section 5.3.

Recall that $T_{out}^{<i>}$ is the timeout period after which node i will be diagnosed as failed by node $n \neq i$ if no broadcast message from node i has been received since the last broadcast. As mentioned earlier, there are two possible scenarios, **S1** and **S2**, that no broadcast message from node i will be received by node n within $T_{out}^{<i>}$. The determination of $T_{out}^{<i>}$ requires to make a tradeoff between these two possibilities, and can thus be formulated as a HT problem with two hypotheses. Specifically, let $Ob(t) \in \{0, 1\}$ indicate whether or not a broadcast message from node i is received within time t, and let T_{nb} be the random variable representing the time to node i's next broadcast. We have two hypotheses:

H_0: **node i is operational** $\quad Ob(t) \sim p_0$,

H_1: **node i is faulty** $\quad Ob(t) \sim p_1$,

where $\sim$ denotes that p_0 and p_1 are the p.d.f. of $Ob(t)$ under the hypothesis H_0 and H_1, respectively. p_0 and p_1 can be expressed as

$$\begin{aligned}
p_0(Ob(t)=0) &= P(\text{no message has been received from node } i \text{ within } t \mid \text{node } i \text{ is operational}) \\
&= P(T_{nb} \geq t \mid \text{node } i \text{ is operational}), \\
p_0(Ob(t)=1) &= P(T_{nb} < t \mid \text{node } i \text{ is operational}) \;=\; 1 - p_0(Ob(t)=0), \\
p_1(Ob(t)=0) &= 1, \quad \text{and} \quad p_1(Ob(t)=1) \;=\; 0.
\end{aligned}$$

Also, the probability that H_0 or H_1 is true without conditioning on any observation can be expressed as $\pi_0 = e^{-\lambda_F t}$ or $\pi_1 = 1 - e^{-\lambda_F t}$, respectively.

Now, a decision $\delta(Ob(t)) \in \{0,1\}$ must be made on which hypothesis must be accepted based on the observation $Ob(t)$. Two types of error may be encountered: (1) *false-alarm*, or H_0 is falsely rejected, the probability of which is denoted by $P_F(\delta)$; (2) *miss*, or H_1 is falsely denied, the probability of which is denoted by $P_M(\delta)$. The corresponding *detection* probability is $P_D(\delta) = 1 - P_M(\delta)$. A criterion for designing a test for H_0 versus H_1, called the Neyman–Pearson criterion [Poo88], is to place a bound on the false-alarm probability and then to minimize the miss probability subject to this constraint; that is, the Neyman–Pearson design criterion is

$$\max_\delta P_D(\delta) \quad \text{subject to } P_F(\delta) \leq \alpha_{ht}, \tag{6.1}$$

where α_{ht} is the significance level of the hypothesis test. Specifically, let the decision $\delta(\cdot)$ be

$$\delta(Ob(t)) = \begin{cases} 1, & \text{if } \pi_1 \cdot p_1(Ob(t)) \geq \pi_0 \cdot p_0(Ob(t)), \\ 0, & \text{otherwise,} \end{cases} \tag{6.2}$$

where the *maximum a posteriori* (MAP) probability is used to determine whether to accept H_1 or not. Then, $P_F(\delta)$ can be expressed as

$$\begin{aligned}
P_F(\delta) &= P(\text{accept } H_1 \mid H_0 \text{ is true}) \;=\; E_0(\,\delta(Ob(t))\,) \\
&= P_0(\,\pi_1 \cdot p_1(Ob(t)) \geq \pi_0 \cdot p_0(Ob(t))\,) \\
&= \sum_{Ob(t)\in\{0,1\}} P(\,\pi_0 \cdot p_0(Ob(t)) \leq \pi_1 \cdot p_1(Ob(t))\,) \cdot p_0(Ob(t)) \\
&= P(p_0(Ob(t)=0) \leq \frac{\pi_1}{\pi_0}) \cdot p_0(Ob(t)=0),
\end{aligned} \tag{6.3}$$

where $E_0(\cdot)$ and $P_0(\cdot)$ denote the expectation and the probability under H_0, and the last equality comes from $P(\pi_0 \cdot p_0(1) \leq \pi_1 \cdot p_1(1)) = 0$. Similarly, $P_D(\delta)$ can be expressed as

$$\begin{aligned}
P_D(\delta) &= E_1(\,\delta(Ob(t))\,) \;=\; P_1(\pi_1 \cdot p_1(Ob(t)) \geq \pi_0 \cdot p_0(Ob(t))) \\
&= P(p_0(Ob(t)=0) \leq \frac{\pi_1}{\pi_0}).
\end{aligned} \tag{6.4}$$

If the expression of $p_0(Ob(t) = 0) = P(T_{nb} \geq t \mid \text{node } i \text{ is operational})$ can be derived as a function of t, then the best $T_{out}^{<i>}$ under the Neyman–Pearson criterion is the minimum t such that both

$$p_0(Ob(t) = 0) \leq \alpha_{ht} \quad \text{and} \quad p_0(Ob(t) = 0) \leq \frac{\pi_1}{\pi_0} = e^{\lambda_F t} - 1 \tag{6.5}$$

are satisfied, in which case $P_D(\delta) = 1$ and $P_F(\delta) \leq \min\{\alpha_{ht}, e^{\lambda_F t} - 1\}$.

6.4 Derivation of $P(T_{nb} \geq t \mid \text{node } i \text{ is operational})$

We now use the randomization technique [Gra77, GM84, MY84] to calculate $P(T_{nb} \geq t \mid \text{node } i \text{ is operational})$. Since this technique can be applied only to a finite state–space continuous–time Markov chain, we model the state evolution of a node as such. We first describe how the system model is constructed. Then, we derive $P(T_{nb} \geq t \mid$ node i is operational) using the randomization technique.

6.4.1 System Model

The state/CET evolution of a node is modeled as a continuous–time Markov chain $\{X(t), t \geq 0\}$ on a finite state space S. Transitions in the Markov chain are characterized by the generator matrix $Q = (q_{ij})$, where q_{ij}, $0 \leq i, j \leq N$, is the transition rate from state i to state j. The parameters needed in the model are λ_i, $\{p_i(j), 1 \leq j \leq E_{max}\}$, and $\{\hat{p}_i(k), 1 \leq k \leq L_{max}\}$, all of which are estimated on–line by each node i and piggy-backed in region–change broadcasts to the other nodes in its buddy set.

We characterize the CET evolution caused by task acceptance/completion under the non-preemptive MLFS discipline. With a minor modification, our model can also be applied to the case when the loading state is queue length. To construct a continuous–time Markov chain on a finite state space, we approximate the deterministic consumption of CET on node i (at a pace of 1 per unit time) as an Erlang distribution with rate K_{er} and shape parameter K_{er}. The Erlang distribution becomes exact (i.e., deterministic with rate 1) as $K_{er} \to \infty$. We choose K_{er} such that $P(T_{nb} \geq t \mid$ node i is operational) obtained from the corresponding $M^{[i]}/E_{K_{er}}/1$ model is very close to that obtained from $M^{[i]}/E_{K_{er}+1}/1$ model. In Section 6.5.1, $K_{er} \geq 5$ is shown to satisfy the above criterion for all combinations of task attributes studied. Each accepted/queued task contributes $K_{er}m$ service stages with probability $p_i(m)$, $1 \leq m \leq E_{max}$, and each service stage is consumed at (an exponential) rate K_{er}.

Definition of state: The state of node i is defined as $\underline{H} = (H_0; H_1; H_2; ...; H_{L_{max}})$, where $H_j \triangleq h_1^j h_2^j ... h_{j+1}^j$ is a sequence of $j+1$ numbers with $h_k^j \in \{0, \ldots, K_{er}E_{max}\}$ representing the number of service stages contributed by the k–th laxity–j task in the node's queue. H_j can be viewed as a record of all laxity–j tasks currently queued on node i. Since all laxity–j tasks queued on node i must start execution by their laxity, there are at most $j+1$ laxity–j tasks that can be queued on node i (in which case all but, perhaps, the last task require 1 unit of execution time). Moreover, let $c_j \triangleq \sum_{k=1}^{j+1} h_k^j$ denote the total number of service stages contributed by all laxity–j tasks, $last(H_j)$ denote the index of the last nonzero entry in H_j, and

$$L_{\text{now}}(\underline{H}) \triangleq \begin{cases} -1, \quad \text{if } \underline{H} = \underline{0}; \\ \text{minimum } \ell \text{ s.t. } \prod_{j=0}^{\ell-1}(1 - \Delta^{>}(c_j)) \times \Delta^{>}(c_j) = 1, \\ \qquad \text{if } \underline{H} \neq \underline{0}, \text{ and } h_1^j \in \{0\} \cup \{K_{er}m : 1 \leq m \leq E_{max}\}\ \forall j; \\ \text{the only index } \ell \text{ s.t. } h_1^\ell \notin \{0\} \cup \{K_{er}m : 1 \leq m \leq E_{max}\}, \\ \qquad \text{if } \underline{H} \neq \underline{0}, \text{ and } \exists j \text{ s.t. } h_1^j \notin \{0\} \cup \{K_{er}m : 1 \leq m \leq E_{max}\} \end{cases}$$

denote the laxity of the task currently under service, where

$$\Delta^{>(\geq)}(x) \triangleq \begin{cases} 1, & \text{if } x > (\geq) 0, \\ 0, & \text{otherwise.} \end{cases}$$

For example, consider a system model with $L_{max} = 3$, $E_{max} = 2$, and $K_{er} = 4$. $L_{\text{now}}(\ (0; 40; 000; 1000)\ ) = 3$ indicates that the task currently being served has 3 time units of laxity and 1 remaining service stage. $L_{\text{now}}(\ (0; 00; 440; 8000)\ ) = 2$ indicates that the task to be served next is the one with 2 time units of laxity and 4 service stages if there are no new laxity–1 task arrivals before the next state transition.

Under the non-preemptive MLFS discipline, the state $\underline{H}$ has the following properties:

P1: $h_k^j \in \mathcal{N}$ is an integer multiple of K_{er} except for perhaps h_1^j, the number of service stages contributed by the laxity–j task currently under service.

P2: The size of the state space is bounded by $\prod_{i=0}^{L_{max}}(K_{er}E_{max} + 1)(E_{max} + 1)^i$ and thus is finite.

P3: Since a task with laxity j is accepted/queued only if the CET contributed by both the tasks with laxity $\leq j-1$ and the task currently under service is no greater than j units of time, we have $c_j > 0$ only if

$$K_{er}j \geq \sum_{n=0}^{j-1} c_n, \quad \forall j \in [L_{\text{now}}(\underline{H}) + 1, L_{max}],$$

or,

$$K_{er}j \geq \sum_{n=0}^{j-1} c_n + h_1^{L_{\text{now}}(\underline{H})}, \quad \forall j \in [0, L_{\text{now}}(\underline{H}) - 1].$$

Note that $c_{L_{\text{now}}(\underline{H})} > 0$ by the definition of $L_{\text{now}}(\underline{H})$ (except for the case of $\underline{H} = \underline{0}$).

P4: Since every laxity–j task queued on node i must be able to start execution by its laxity, the number of service stages queued "in front of" it must be $\leq K_{er}j$, i.e.,

$$\sum_{n=0}^{j-1} c_n + \sum_{n=1}^{last(H_j)-1} h_n^j \leq K_{er}j, \quad \forall j \in [L_{\text{now}}(\underline{H}) + 1, L_{max}],$$

or,

$$\sum_{n=0}^{j-1} c_n + h_1^{L_{\text{now}}(\underline{H})} + \sum_{n=1}^{last(H_j)-1} h_n^j \leq K_{er}j, \quad \forall j \in [0, L_{\text{now}}(\underline{H}) - 1].$$

For example, consider again the system model with $L_{max} = 3$, $E_{max} = 2$, and $K_{er} = 4$. The state (0;10;440;8000) is allowed, while (0;10;480;$\underline{4}$000) is not, because the task with 3 time units of laxity and 4 service stages (represented by the underlined number 4) in the latter state violates **P3** and **P4**. The state (0;48;000;8000) is allowed, while (0;48;000;1000) is not, because in the latter state the task with 3 time units of laxity (represented by 1) is currently in service, and thus, the task with 1 time unit of laxity and 8 service stages (represented by 8) cannot be queued.

As indicated in **P2**, the size of the state space is bounded and is actually much less than the given bound because of **P1** and **P3–P4**. It, however, grows significantly as L_{max} or E_{max} or K_{er} increases, but as will be clearer later in this section, the generator matrix Q of the corresponding Markov chain is very sparse, so one can exploit the sparseness of Q — e.g., use the modified SERT algorithm proposed in [GM84] — to economically store sparse matrices, and to alleviate the computational difficulty.

Determination of transition rates: There are two task activities that cause state transitions: one is task acceptance by node i, and the other is CET consumption by node i. The task transfers resulted from the acceptance of a newly–arrived task under the MLFS scheduling discipline are figured in task acceptance. (Recall that some tasks originally queued on the node may have their laxities missed as a result of inserting a newly–arrived task into the sorted task queue, and must thus be transferred out.)

A. The transition caused by task acceptance: Assume that the system is in state $\underline{H}$, and will make a transition to state $\underline{H}'_{\ell,K_{er}m} \triangleq (H'_0; H'_1; ...; H'_\ell; ...; H'_{L_{max}})$ upon acceptance of a task with laxity ℓ and execution time m, where $1 \leq \ell \leq L_{max}$ and $1 \leq m \leq E_{max}$. Then

(1) $c'_j = c_j$ (or equivalently, $H'_j = H_j$), $1 \le j \le \ell - 1$, i.e., the CET contributed by tasks with laxity $\le \ell - 1$ will not be affected by the acceptance of a task with laxity ℓ;

(2) H'_j equals H_j, perhaps with the last few entries ($\ell + 1 \le j \le L_{max}$) replaced by 0 (so $c'_j \le c_j$). That is, the tasks originally queued with laxity $> \ell$ may have to be transferred out because of the insertion of a newly–arrived task.

(3) The number of nonzero entries in H_ℓ is not greater than ℓ, and $H'_\ell = h^\ell_1 ... h^\ell_{last(H_\ell)}\ K_{er}m\ 0...0$, i.e., H'_ℓ consists of the nonzero entries in H_ℓ followed by the number $K_{er}m$ (and possibly a few 0's to make the number of entries equal to $\ell + 1$).

(4) The corresponding transition rate (under the non–preemptive policy) is

$$q_{\underline{H},\underline{H}'_{\ell,K_{er}m}} = \lambda_i \hat{p}_i(\ell) p_i(m) \cdot \text{Check\_Cet}(\ell) \cdot \prod_{\substack{t=\ell+1 \\ t \notin zero(\underline{C})}}^{L_{max}} \Big\{ comp(H_t, H'_t) \cdot \text{Task\_Not\_Transfer}(t) + (1 - comp(H_t, H'_t)) \cdot \text{Task\_Transfer}(t) \Big\}, \tag{6.6}$$

where

$$\text{Check\_Cet}(\ell) \triangleq \begin{cases} \Delta \ge (K_{er}\ell - \sum_{j=0}^{\ell} c_j), & \text{if } L_{\text{now}}(\underline{H}) \le \ell, \\ \Delta \ge (K_{er}\ell - (\sum_{j=0}^{\ell} c_j + h_1^{L_{\text{now}}(\underline{H})})), & \text{if } L_{\text{now}}(\underline{H}) > \ell; \end{cases}$$

$$zero(\underline{C}) \triangleq \text{the set of indices } j \text{ such that } c_j = 0;$$

$$comp(H_t, H'_t) \triangleq \begin{cases} 1, & \text{if } H_t = H'_t, \\ 0, & \text{otherwise;} \end{cases}$$

$$\text{Task\_Not\_Transfer}(t) \triangleq \begin{cases} 1, & \text{if } t = L_{\text{now}}(\underline{H}) \text{ and } last(H_t) = 1, \\ \Delta \ge (K_{er}t - (\sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H_t)-1} h^t_j)), & \text{if } t = L_{\text{now}}(\underline{H}) \text{ and } last(H_t) > 1, \\ \Delta \ge (K_{er}t - (\sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H_t)-1} h^t_j + h_1^{L_{\text{now}}(\underline{H})})), & \text{if } t < L_{\text{now}}(\underline{H}), \\ \Delta \ge (K_{er}t - (\sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H_t)-1} h^t_j)), & \text{if } t > L_{\text{now}}(\underline{H}); \end{cases}$$

$$\text{Task\_Transfer}(t) \triangleq \begin{cases} \Delta \ge (K_{er}t - (\sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H'_t)-1} h^t_j)) \times \\ \quad \Delta > ((\sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H'_t)} h^t_j) - K_{er}t), & \text{if } t = L_{\text{now}}(\underline{H}) \text{ and } last(H'_t) \ge 2, \\ \Delta > (\sum_{j=0}^{t-1} c'_j + h^t_1 - K_{er}t), & \text{if } t = L_{\text{now}}(\underline{H}) \text{ and } last(H'_t) = 1, \\ 0, & \text{if } t = L_{\text{now}}(\underline{H}) \text{ and } last(H'_t) = 0, \\ \Delta \ge (K_{er}t - (\sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H'_t)-1} h^t_j)) \times \\ \quad \Delta > ((\sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H'_t)} h^t_j) - K_{er}t), & \text{if } t > L_{\text{now}}(\underline{H}) \text{ and } last(H'_t) \ne 0, \\ \Delta > (\sum_{j=0}^{t-1} c'_j - K_{er}t), & \text{if } t > L_{\text{now}}(\underline{H}) \text{ and } last(H'_t) = 0, \\ \Delta \ge (K_{er}t - (\sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H'_t)-1} h^t_j + h_1^{L_{\text{now}}(\underline{H})})) \times \\ \quad \Delta > ((\sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H'_t)} h^t_j) + h_1^{L_{\text{now}}(\underline{H})} - K_{er}t), & \text{if } t < L_{\text{now}}(\underline{H}) \text{ and } last(H'_t) \ne 0, \\ \Delta > (\sum_{j=0}^{t-1} c'_j + h_1^{L_{\text{now}}(\underline{H})} - K_{er}t), & \text{if } t < L_{\text{now}}(\underline{H}) \text{ and } last(H'_t) = 0; \end{cases}$$

The physical meanings of Eq. (6.6) are given below:

(a) The first factor $\lambda_i \hat{p}_i(\ell) p_i(m)$ is the arrival rate of tasks with ℓ time units of laxity and m units of execution time on node i.

(b) The second factor Check_Cet(ℓ) accounts for the fact that a newly-arrived task with laxity ℓ will be queued/accepted on node i only if one of the following two conditions holds: (i) the CET contributed by tasks with laxity $\leq \ell$ is less than or equal to ℓ, i.e., $K_{er}\ell \geq \sum_{j=0}^{\ell} c_j$, if the laxity of the currently executing task $\leq \ell$; or (ii) the CET contributed by the tasks with laxity $\leq \ell$ and the task currently under service is $\leq \ell$ if the laxity of the currently executing task $> \ell$ (i.e., no preemption).

(c) The last factor accounts for the possible task transfers caused by the acceptance of the arrived task. Since only tasks with laxity $> \ell$ will be affected by the insertion of the newly-arrived task with laxity ℓ, $\prod$ is performed from $t = \ell + 1$ to $t = L_{max}$ except for those t's with $c_t = 0$. The transition could occur with rate $\lambda_i \hat{p}_i(\ell) p_i(m)$ if, in addition to the conditions in Check_Cet(ℓ), one of the following conditions holds, $\forall t \in [\ell + 1, L_{max}]$:

- **(i)** $H_t = H'_t$ and all tasks queued with laxity t can still be completed in time after the insertion of the arrived task, i.e., $K_{er}t \geq \sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H_t)-1} h^t_j$ if the laxity of the currently executing task $\leq t$, or, $K_{er}t \geq \sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H_t)-1} h^t_j + h^t_1 \mathrm{now}(\underline{H})$ if the laxity of the currently executing task $> t$.
- **(ii)** H'_t equals H_t except with the last $(last(H_t) - last(H'_t))$ entries replaced by zero, i.e., a number $(last(H_t) - last(H'_t))$ of tasks with laxity t must be transferred out. For example, if $t > L_{\mathrm{now}}(\underline{H})$, exactly i tasks with laxity t have to be transferred out if and only if both $\sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H_t)-i-1} h^t_j \leq K_{er}t$ and $\sum_{j=0}^{t-1} c'_j + \sum_{j=1}^{last(H_t)-i} h^t_j > K_{er}t$ hold. The cases with $t < L_{\mathrm{now}}(\underline{H})$ and $t = L_{\mathrm{now}}(\underline{H})$ can be similarly reasoned about.

B. The transitions caused by CET consumption: The deterministic consumption of CET at a pace of 1 per unit time is approximated as a K_{er}-Erlang distribution with rate K_{er}. Besides, at the end of each time unit (i.e., at the end of every K_{er} service stages), all laxities have to be decremented by 1 to account for the fact that the laxity of a task is measured w.r.t. the current time. Specifically, the system makes a transition from $\underline{H}$ to $\underline{H}'_{\ell,-1} = (H'_0; H'_1; ...; H'_\ell; ...; H'_{L_{max}})$, with transition rate

$$q_{\underline{H},\underline{H}'_{\ell,-1}} = \begin{cases} K_{er}, & \text{if } \ell = L_{\mathrm{now}}(\underline{H}), \\ 0, & \text{otherwise,} \end{cases} \tag{6.7}$$

where

(1) if $(h_1^\ell - 1) \notin \{K_{er}m : 1 \le m \le E_{max}\} \cup \{0\}$,

$$H'_\ell \;=\; (h_1^\ell - 1)h_2^\ell ... h_{\ell+1}^\ell, \text{ and } H'_j \;=\; H_j \;\; \forall j \neq \ell.$$

(2) if $(h_1^\ell - 1) \in \{K_{er}m : 1 \le m \le E_{max}\} \cup \{0\}$,

$$H'_{L_{max}} \;=\; \underline{0}, \quad H'_j \;=\; H_{j+1} \;=\; h_1^{j+1} ... h_{j+1}^{j+1}, \;\; \forall j \in [0, \ell-2] \cup [\ell, L_{max}-1],$$

$$H'_{\ell-1} \;=\; \begin{cases} (h_1^\ell - 1)h_2^\ell ... h_\ell^\ell, & \text{if } h_1^\ell > 1, \\ h_2^\ell ... h_\ell^\ell 0, & \text{if } h_1^\ell = 1. \end{cases}$$

The last nonzero transition rate is

$$q_{\underline{H},\underline{H}} = -(\sum_{\ell,m} q_{\underline{H},\underline{H}'_{\ell,K_{er}m}} + \sum_{\ell} q_{\underline{H},\underline{H}'_{\ell,-1}}) \triangleq -q_{\underline{H}}. \tag{6.8}$$

The model constructed above is a continuous-time Markov chain, because (1) the residence time at each state is exponentially distributed, and (2) the next state the system will visit depends only on the current state and the task acceptance/completion activities occurred during the residence at the current state. The sparseness of Q comes from the fact that all the other entries (except for the transition rates in Eq. (6.6)–(6.8)) in Q are zero. For example, the only possible transitions from state (0;10;400;4800) in the system model with $L_{max} = 3$, $E_{max} = 2$, and $K_{er} = 4$ are to (0;40;480;0000), (0;14;400;4000), (0;18;000;4000), (0;10;440;4000), (0;10;480;0000), and (0;10;400;4800) with transition rate K_{er}, $\lambda_i p_i(1)\hat{p}_i(1)$, $\lambda_i p_i(2)\hat{p}_i(1)$, $\lambda_i p_i(1)\hat{p}_i(2)$, $\lambda_i p_i(2)\hat{p}_i(2)$, and $-(K_{er} + \sum_{1 \le \ell,m \le 2} \lambda_i p_i(m)\hat{p}_i(\ell))$, respectively. The transition (0;10;400;4800) $\rightarrow$ (0;10;400;484$\underline{0}$) is not possible, because the newly-arrived task with laxity $\ell = 3$ (represented by the underlined 4) will not be accepted (i.e., Check_Cet(ℓ) = 0, because $\sum_{j=0}^{\ell} c_j > K_{er}\ell$). The transition (0;10;400;4800) $\rightarrow$ (0;10;480;$\underline{4}$000) is not possible either, because the task queued with 3 time units of laxity and 4 service stages (represented by the underlined 4) must also be transferred (in addition to the task with 3 units of laxity and 8 service stages) after inserting the newly-arrived task. Similarly, the only possible transitions from state (0;40;000;1400) are to (4;00;400;0000), (0;40;400;1400), (0;40;800;1000), (0;40;000;1440), (0;40;000;1480), and (0;40;000;1400) with transition rate K_{er}, $\lambda_i p_i(1)\hat{p}_i(2)$, $\lambda_i p_i(2)\hat{p}_i(2)$, $\lambda_i p_i(1)\hat{p}_i(3)$, $\lambda_i p_i(2)\hat{p}_i(3)$, and $-(K_{er} + \sum_{\substack{1 \le m \le 2 \\ 2 \le \ell \le 3}} \lambda_i p_i(m)\hat{p}_i(\ell))$, respectively. The transition (0;40;000;1400) $\rightarrow$ (0;4$\underline{4}$;000;1400) is not possible, because the task currently under service has laxity 3 (i.e., L_{now} ((0; 40; 000; 1400)) = 3), and the newly-arrived task with $\ell = 1$ (represented by the underlined 4) will not be accepted under a non-preemptive policy (i.e., $\sum_{j=0}^{\ell} c_j + h_1^{L_{\text{now}}(\underline{H})} > K_{er}\ell$).

6.4.2 Probability Calculation with the Randomization Technique

We now use the randomization technique to calculate the probability that a node does not broadcast any message in $[0,t]$, given it is operational in $[0,t]$. This technique was introduced in [Gra77, GM84, MY84] as a method for computing transient probabilities of Markov processes with finite state spaces [HS93a], and is summarized in the Appendix B.

Recall that in the proposed LS mechanism, a node's states are divided into K_t disjoint subsets: $[0,TH_1]$, $(TH_1,TH_2]$, ..., (TH_{K_t-1},∞), where $TH_k, 1 \leq k \leq K_t - 1$ are the thresholds of the node's CET. A node will broadcast to other nodes its change of state region whenever its state/CET crosses even-numbered thresholds, TH_{2j}, $1 \leq j \leq \lceil \frac{K_t}{2} \rceil - 1$. We thus define $S_j = \{\underline{H} : K_{cr} \cdot TH_{2(j-1)} \leq \sum_{n=1}^{L_{max}} (\sum_{k=1}^{n+1} h_k^n) < K_{cr} \cdot TH_{2j}\}$ as the j-th broadcast state region, where $TH_0 \triangleq 0$, the expression $\sum_{k=1}^{n+1} h_k^n$ is the number of service stages contributed by laxity-n tasks (i.e., c_n), and the expression between inequalities $\sum_{n=1}^{L_{max}} (\sum_{k=1}^{n+1} h_k^n)$ is simply the total number of service stages queued on the node.

Let $r_j(n,k), 0 \leq k \leq n+1$, be the probability that the discrete-time Markov chain, Y, obtained after the randomization of $X(t)$ visits k times the states in S_j out of n state changes. For example, $r_j(n,n+1)$ is the probability that Y always stays in S_j while there are n state changes. Then, $P(T_{nb} \geq t \mid$ node i is operational and was in S_j during the last broadcast), $1 \leq j \leq \lceil \frac{K_t}{2} \rceil$, is the probability that the underlying Markov chain always stays in S_j, no matter how many state changes have occurred in $[0,t]$. Thus,

$$\begin{aligned} & P(T_{nb} \geq t \mid \text{node } i \text{ is operational and was in } S_j \text{ during the last broadcast}) \\ &= \sum_{n=0}^{\infty} r_j(n,n+1) \cdot P(n \text{ state changes in time } t) \\ &= \sum_{n=0}^{\infty} r_j(n,n+1) \cdot e^{-\Lambda t}(\Lambda t)^n/n! \end{aligned}$$

where Λ is the rate of the Poisson process obtained after the randomization.

The error, e_m, resulting from the truncation of the infinite sum in the above equation can be easily bounded as

$$e_m = \sum_{n=m+1}^{\infty} e^{-\Lambda t}(\Lambda t)^n/n! \cdot r_j(n,n+1) \leq 1 - \sum_{n=0}^{m} e^{-\Lambda t}(\Lambda t)^n/n!. \tag{6.9}$$

The $\leq$ in Eq. (6.9) results from the inequality $r_j(n,n+1) \leq 1$. The value of m can be determined *a priori* for any given error tolerance.

$r_j(n,k)$ (and $r_j(n,n+1)$, in particular) can be easily calculated using the recursive approach proposed in [dSeSG86] (and later studied in depth in [dSeSG89]). That is, let $r_j(n,k,\underline{H})$ be the probability that the underlying Markov chain Y are k times in S_j out of

n state changes and the state visited in the last transition is state $\underline{H}$. $r_j(n,k,\underline{H})$ depends on

- $r_j(n-1,k-1,\hat{\underline{H}})$, $\forall\hat{\underline{H}} \in S$, if $\underline{H} \in S_j$, since we have to increment the number of states $\in S_j$ visited by one for the previous state change from $\hat{\underline{H}}$ to $\underline{H}$;
- $r_j(n-1,k,\hat{\underline{H}})$ $\forall\hat{\underline{H}} \in S$, if $\underline{H} \notin S_j$, since the number of states $\in S_j$ visited remains the same for the current state change from $\hat{\underline{H}}$ to $\underline{H}$.

So,

$$r_j(n,k,\underline{H}) = \begin{cases} \sum_{\hat{\underline{H}}\in S} r_j(n-1,k-1,\hat{\underline{H}}) \cdot \mathcal{P}_{\hat{\underline{H}},\underline{H}}, & \text{if } \underline{H} \in S_j, \\ \sum_{\hat{\underline{H}}\in S} r_j(n-1,k,\hat{\underline{H}}) \cdot \mathcal{P}_{\hat{\underline{H}},\underline{H}}, & \text{if } \underline{H} \notin S_j, \end{cases} \tag{6.10}$$

where $\mathcal{P}$ is the transition matrix of Y, and the initial conditions are

$$\begin{aligned} r_j(0,1,\underline{H}) &= \begin{cases} 1, & \text{if } \underline{H} \in S_j \text{ and } \underline{H} \text{ is the state representation of } \underline{T_Q}, \\ 0, & \text{otherwise,} \end{cases} \\ r_j(0,0,\underline{H}) &= 0, \end{aligned} \tag{6.11}$$

where Eq. (6.11) comes from the fact that given the CET was in S_j during the last broadcast, the node must be initially in a state $\in S_j$, and the k within the expression of $r_j(n,k,\underline{H})$ must be ≥ 1. Finally, $r_j(n,k) = \sum_{\underline{H}\in S} r_j(n,k,\underline{H})$.

Since we are interested in obtaining $r_j(n,n+1)$, we need only to compute $r_j(n,n+1,\underline{H}), \forall\underline{H} \in S_j$, as $r_j(n,n+1,\underline{H}) = 0$, $\forall\underline{H} \notin S_j$. Thus, Eq. (6.10) reduces to

$$r_j(n,n+1,\underline{H}) = \sum_{\hat{\underline{H}}\in S_j} r_j(n-1,n,\hat{\underline{H}}) \cdot \mathcal{P}_{\hat{\underline{H}},\underline{H}} \quad \forall\underline{H} \in S_j.$$

6.5 Numerical Examples

The proposed timeout mechanism is evaluated in the following sequence:

1. Discussion on the parameters considered/varied in performance evaluation.
2. Discussion on $T_{out}^{<i>}$ (a) w.r.t. task attributes, and (b) w.r.t. the state in which a node was during its latest broadcast.
3. Performance evaluation: First, we comparatively evaluate (a) LS with no timeout mechanism, (b) LS with fixed timeouts, (c) LS with the calculated best timeouts, and (d) LS with immediate detection of each node failure upon its occurrence. Second, we study the negative impact of statistical fluctuation in external task arrivals on the proposed LS mechanism.

6.5.1 Parameters Considered/Varied

Both 16-node and 64-node regular systems are used in our simulations. Both node failure and recovery rates are assumed to be exponential with λ_F varying from 10^{-2} to 10^{-4} and μ_F being fixed at 10^{-1}. The size of the buddy set, the tunable parameters of the proposed LS mechanism, the computational overheads assumed, the ranges varied for task parameters (e.g., the average external task arrival rate per node, $\overline{\lambda_{ext}}$, the ratio of $\frac{e_{j+1}}{e_j}$, and the ratio of $\frac{\ell_{j+1}}{\ell_j}$), the confidence level achieved (in simulation), and the notation used all conform to those described in Section 3.5.

The transmission delay associated with each task transfer is varied from 10% to 50% of the execution time of each task being transferred. The broadcast-message-transmission delay is assumed to be negligible.[3] The medium-queueing delay which is experienced by both broadcast messages and transferred tasks and which dynamically changes with system load and traffic is modeled as a linear function of the number of tasks/messages queued for the particular medium. The shape parameter K_{er} is chosen to be 5, since $P(T_{nb} \geq t \mid \text{node is operational})$ thus derived is almost indistinguishable from that derived with $K_{er} \geq 6$ (Fig. 6.2).

The case with hyperexponential interarrival times represents a system potentially with bursty task arrivals. The squared coefficient of variation of hyperexponential arrivals (CV^2) is varied from 1 to 91. Again, we present only those results that we believe are the most relevant, interesting, and/or representative.

6.5.2 Discussion of $T_{out}^{<i>}$

$T_{out}^{<i>}$ increases as $p_0(Ob(t) = 0) = P(T_{nb} \geq t \mid \text{node } i \text{ is operational})$ for a given t increases. Figs. 6.3 – 6.5 illustrate how $p_0(Ob(t) = 0)$ (and thus, $T_{out}^{<i>}$) varies markedly with the task arrival rate, the state of node i at the time of its latest broadcast, and the length of broadcast intervals, respectively.

As the composite task arrival rate increases, a node tends to cross its broadcast thresholds more often if there is a threshold nearby and to the right of the node's current state. Thus, in Fig. 6.3, the increase in λ_i yields a smaller $p_0(Ob(t) = 0)$ for a given t (e.g., the more likely a broadcast message is issued within time t). Similarly, as evidenced in the curves labeled as "initial state=2.0" in Fig. 6.4 or in the curves labeled as "init. state=5.0" in Fig. 6.5, the closer the initial state of a node is to a broadcast threshold, the more likely

[3]The physical transfer of the virtual memory image of a task may require tens of communication packets, while a region-change broadcast would in all likelihood need at most one packet.

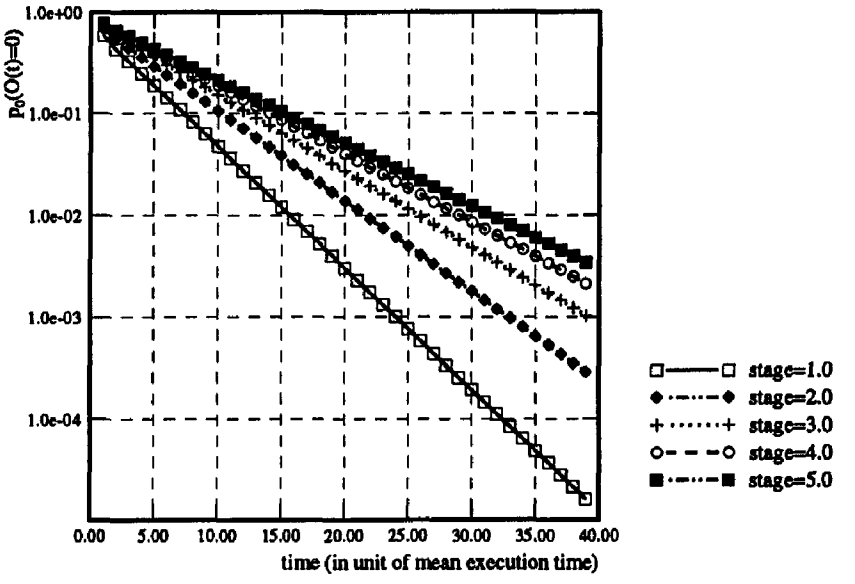


Figure 6.2: $P(T_{nb} \geq t \mid \text{node } i \text{ is operational})$ derived w. r. t. shape parameter K_{er}. $\lambda_i = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$ (mean $ET = 1.0$), and $L = \{1.0, 2.0, 3.0\}_{\frac{1}{3}}$. Node i has 4 state regions with each interval equal to 1 (except for the last interval), i.e., $S_1 = [0, 2.0]$, $S_2 = (2.0, \infty)$. The state of node i is CET=1.0 in the last broadcast.

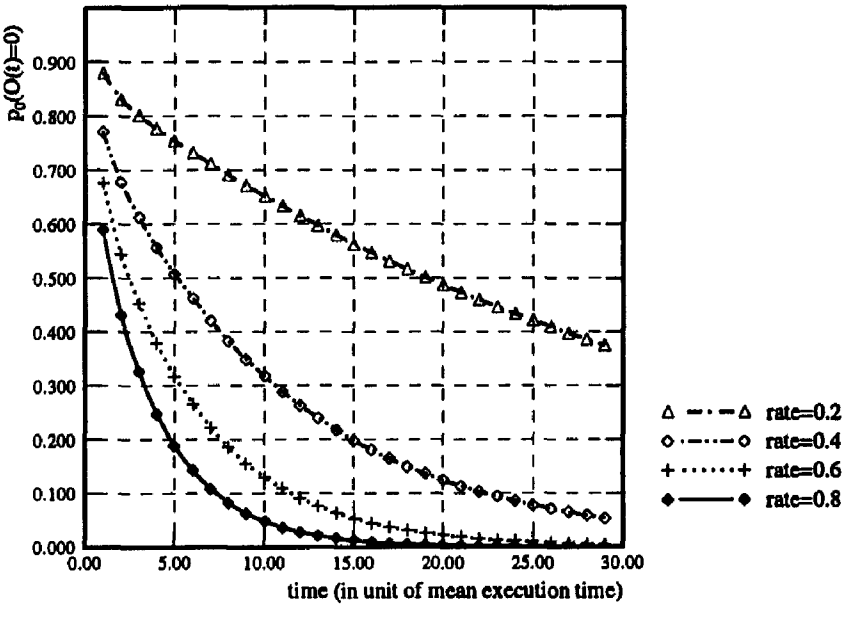


Figure 6.3: $P(T_{nb} \geq t \mid \text{node } i \text{ is operational})$ w. r. t. task arrival rate λ_i. $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1.0, 2.0, 3.0\}_{\frac{1}{3}}$, and $K_{er} = 5$. Node i has 4 state regions determined by $TH_1 = 1.0$, $TH_2 = 2.0$, and $TH_3 = 3.0$. $S_1 = [0, 2.0]$, $S_2 = (2.0, \infty)$. The state of node i is CET=1.0 in the last broadcast.

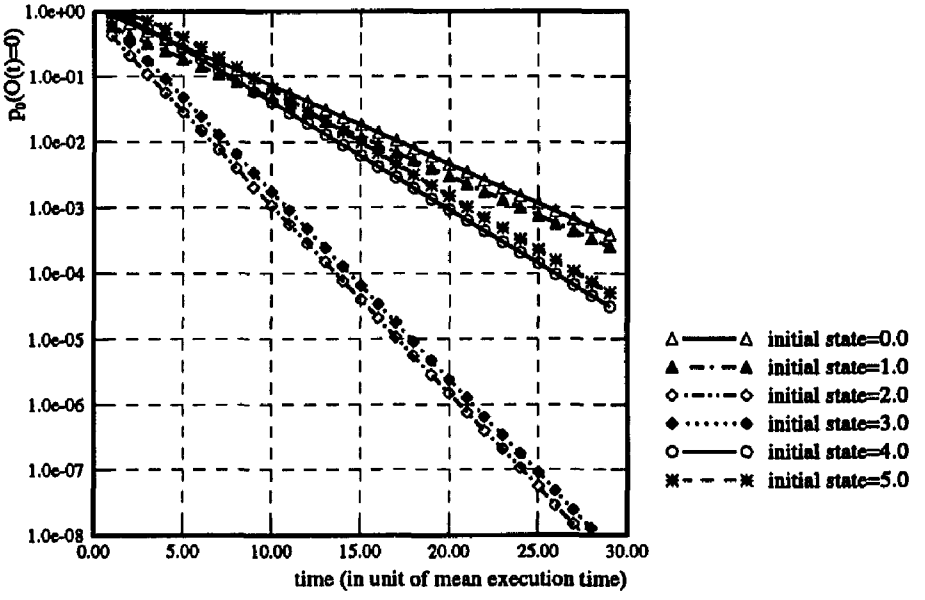


Figure 6.4: $P(T_{nb} \geq t \mid \text{node } i \text{ is operational})$ w. r. t. the initial state node i is in. $\lambda = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1.0, 2.0, 3.0\}_{\frac{1}{3}}$, and $K_{er} = 5$. Node i has 4 state regions determined by $TH_1 = 1.0$, $TH_2 = 2.0$, and $TH_3 = 3.0$. $S_1 = [0, 2.0]$, $S_2 = (2.0, \infty)$.

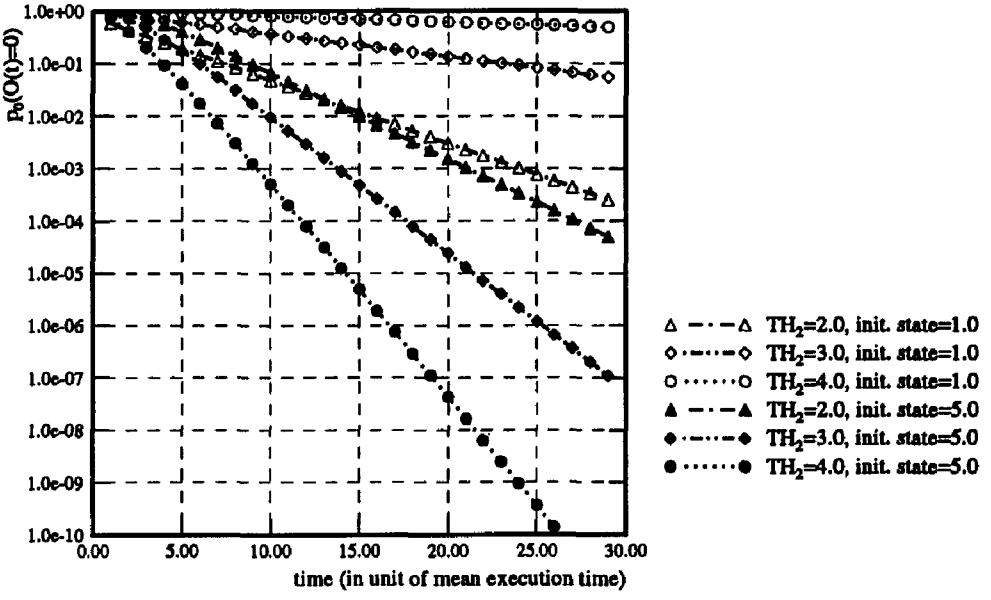


Figure 6.5: $P(T_{nb} \geq t \mid \text{node } i \text{ is operational})$ w. r. t. the length of broadcast interval. $\lambda = 0.8$, $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1.0, 2.0, 3.0\}_{\frac{1}{3}}$, and $K_{er} = 5$. Node i has 4 state regions determined by $TH_1 = \frac{1}{2}TH_2$, TH_2, and $TH_3 = \frac{3}{2}TH_2$.

the node's state will cross the threshold, thus increasing the possibility of a region–change broadcast.

One interesting observation in Fig. 6.4 is that the probability of a node with initial state 4.0–5.0 broadcasting within time t is smaller than that of a node with initial state 0.0–1.0, when $t \leq 5$ (units of mean execution time), but becomes greater when $t > 5$. This is because a node with initial state 4.0–5.0 will not accept most of its arrived tasks and tends to consume its CET. On the other hand, task acceptance and completion activities may alternate on a node with initial state 0.0–1.0. Consequently, it is more likely for a node with initial state 4.0–5.0 to reach the broadcast threshold after it consumes all its CET (e.g., after 5 units of mean execution time).

Fig. 6.5 also demonstrates how the size of each broadcast interval affects $p_0(Ob(t) = 0)$. As shown in the curves labeled "init. state=1.0" in Fig. 6.5, the larger the broadcast interval is, the less likely a node's state will cross any broadcast threshold, thus resulting in a higher $p_0(Ob(t) = 0)$ for a given t.

Since $p_0(Ob(t))$ varies drastically with the task attributes and the initial state of a node, the on–line calculation of $T_{out}^{<i>}$ is very important to the design of a timeout mechanism. Tables 6.1 and 6.2 give some numerical values of $T_{out}^{<i>}$ for different task attributes, confidence intervals α_{ht}, and node failure rates λ_F.

6.5.3 Performance Evaluation

We use the probability of dynamic failure P_{dyn} and the probability of false alarm P_F (the probability of falsely diagnosing a healthy node as failed) as the performance measures. Since each node refrains itself from sending tasks to the falsely–diagnosed nodes (as well as to the truly failed nodes), P_F is a measure in incorrectly limiting the LS capacity of a system. That is, a larger P_F will leave a node with fewer candidate nodes for task transfers, thus deteriorating the LS performance.

Using trace–driven simulations, we comparatively evaluate the performance improvement achievable with the on–line calculated best timeout mechanism. We compare the proposed LS mechanism with the best timeout period against the case of using a fixed timeout period where a node n (1) considers node i failed if it has not heard from node i for $T_{fixed}^{<i>}$ and (2) broadcasts its fault–free status if it has been silent for $T_{fixed}^{<i>}$, where $T_{fixed}^{<i>}$ is a constant selected independently of node i's task attributes and state. We also compare the proposed timeout mechanism with two baseline mechanisms. The first baseline assumes no timeout mechanism, while the second is an ideal case where (1) each node immediately detects the failure of another node upon its occurrence and (2) no false alarm occurs.

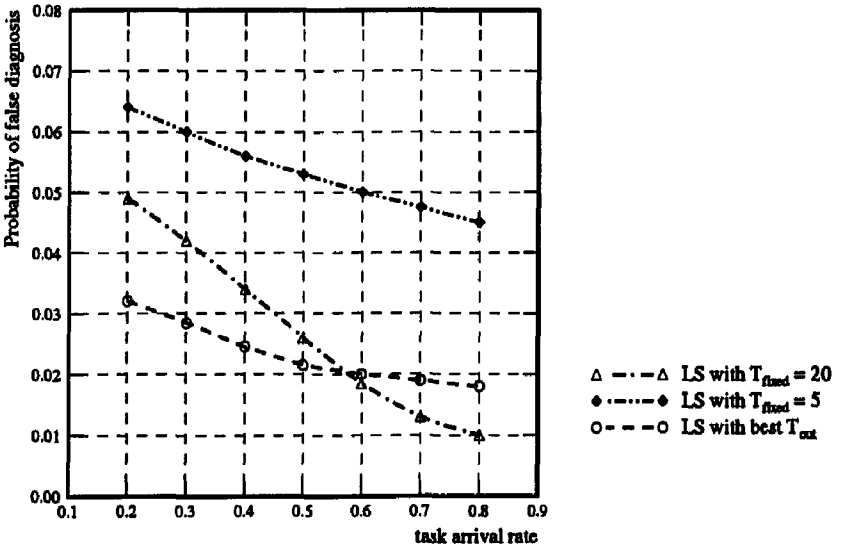


Figure 6.6: Performance comparison w.r.t. P_F for different timeout periods in a 16-node ($K_{np} = 16$) system. External (local) task attributes for node i: $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1, 2, 3\}_{\frac{1}{3}}$, and $K = 5$. $\lambda_F = 10^{-2}$, $\mu_F = 0.1$, and $\alpha_{ht} = 0.05$. The task-transfer delay is assumed to be 10% of the execution time of the transferred task. Each node has 4 state regions determined by $TH_1 = 1.0$, $TH_2 = 2.0$, and $TH_3 = 3.0$. $S_1 = [0, 2.0]$, $S_2 = (2.0, \infty)$.

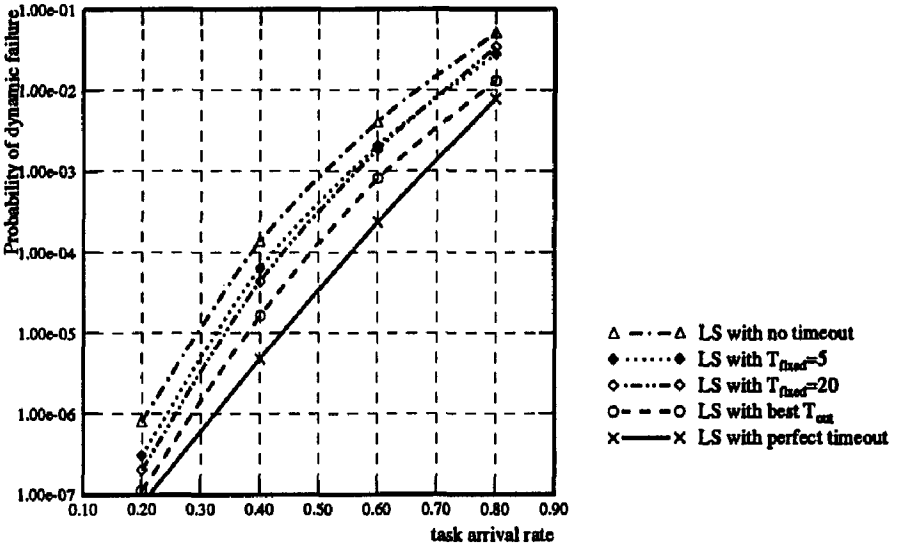


Figure 6.7: Performance comparison w.r.t. P_{dyn} for different timeout periods in a reliable 16-node ($K_{np} = 16$) system. Local task attributes for node i: $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1.0, 2.0, 3.0\}_{\frac{1}{3}}$, and $K = 5$. $\lambda_F = 10^{-3}$, $\mu_F = 0.1$, and $\alpha_{ht} = 0.05$. The task-transfer delay is assumed to be 10% of the execution time of the transferred task. Each node has 4 state regions determined by $TH_1 = 1.0$, $TH_2 = 2.0$, and $TH_3 = 3.0$. $S_1 = [0, 2.0]$, $S_2 = (2.0, \infty)$.

Distribution of Task Laxity	Initial State	Best T^*_{out}			
		$\alpha_{ht} = 0.20$, $\lambda_F = 10^{-2}$	$\alpha_{ht} = 0.05$, $\lambda_F = 10^{-2}$	$\alpha_{ht} = 0.05$, $\lambda_F = 10^{-3}$	$\alpha_{ht} = 0.01$, $\lambda_F = 10^{-3}$
$\{1\}_1$	0	9.1	11.4	15.6	17.2
	1.0	8.0	9.9	14.4	15.7
	2.0	3.4	3.4	5.3	5.3
	3.0	4.8	4.8	7.1	7.1
$\{1,2,3\}_{1/3}$	0	9.1	11.4	15.6	17.2
	1.0	8.0	9.9	14.4	15.7
	2.0	5.8	6.2	10.4	10.5
	3.0	7.2	8.2	12.0	12.5
	4.0	8.1	9.5	13.1	13.8
	5.0	9.1	10.7	14.1	15.0
$\{1,2,3,4,5\}_{\{0.1,0.1,0.2,0.3,0.3\}}$	0	9.1	11.4	15.6	17.2
	1.0	8.0	9.9	14.4	15.7
	2.0	7.9	10.4	16.3	18.9
	3.0	10.3	14.4	19.3	22.8
	4.0	11.8	16.6	21.1	25.1
	5.0	11.8	18.1	22.3	26.6
	6.0	13.6	19.2	23.2	27.7

Table 6.1: Best timeout periods w. r. t. the initial state, α_{ht}, and λ_F. $\lambda_i = 0.8$, and $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$. Node i has 4 state regions determined by $TH_1 = 1.0$, $TH_2 = 2.0$, and $TH_3 = 3.0$. $S_1 = [0, 2.0]$, $S_2 = (2.0, \infty)$.

Fig. 6.6 and Figs. 6.7 –6.9 plot the curves of P_F and the curves of P_{dyn} for LS with different timeout periods w.r.t. different combinations of λ_F and system size K_{np}, respectively. From these figures, we make the following observations:

- In general, P_F decreases as (1) task arrivals/transfers get more frequent (i.e., as the system load increases), and (2) the timeout periods get larger. Thus, in Fig. 6.6 the case of $T^{<i>}_{fixed} = 20$ performs best w.r.t. P_F for medium to heavy system loads ($\lambda^{ext}_i \geq 0.6$, where $5 < T^{<i>}_{out} < 20$ as listed in Table 6.2). For light to medium system loads ($0.2 \leq \lambda^{ext}_i < 0.6$), the case of $T^{<i>}_{out}$ performs best w.r.t. P_F, because usually $T^{<i>}_{out} > 20$ (Table 6.2).
- The assumed 5% chance of incorrect diagnosis ($\alpha_{ht} = 0.05$ in the HT formulation) is reduced with a few extra, timely messages broadcast by each node to inform other nodes of its fault-free status after a silence for $T^{<i>}_{out}$.
- The case with the on-line calculated $T^{<i>}_{out}$ outperforms all the other fixed–timeout cases tested in reducing P_{dyn} over a wide range of system load. The case with $T^{<i>}_{fixed} = 20$ is inferior to that with $T^{<i>}_{out}$ for medium to heavy system loads due to its

λ	L	Initial state	Best T_{out}
0.4	$\{1\}_1$	0	32.6
		1.0	29.8
		2.0	3.4*
		$\geq$ 3.0	$\geq$ 4.8*
	$\{1,2,3\}_{1/3}$	0	32.6
		1.0	29.8
		2.0	4.6*
		3.0	6.4
		4.0	7.8
		$\geq$ 5.0	$\geq$ 9.1
	$\{1,2,3,4,5\}_{\{0.1,0.1,0.2,0.3,0.3\}}$	0	32.6
		1.0	29.8
		2.0	5.5
		3.0	8.4
		4.0	10.3
		5.0	11.8
		$\geq$ 6.0	$\geq$ 13.0
0.8	$\{1\}_1$	0	11.4
		1.0	9.9
		2.0	3.4*
		$\geq$ 3.0	$\geq$ 4.8*
	$\{1,2,3\}_{1/3}$	0	11.4
		1.0	9.9
		2.0	6.2
		3.0	8.2
		4.0	9.5
		$\geq$ 5.0	$\geq$ 10.7
	$\{1,2,3,4,5\}_{\{0.1,0.1,0.2,0.3,0.3\}}$	0	11.4
		1.0	9.9
		2.0	10.4
		3.0	14.4
		4.0	16.6
		5.0	18.1
		$\geq$ 6.0	$\geq$ 19.2

(in units of mean task execution time.
* indicates $e^{\lambda_F t} - 1$ dominates the determination of T_{out}.)

Table 6.2: Best timeout periods w. r. t. the task characteristic and the initial state of node i. $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$. $\alpha_{ht} = 5 \times 10^{-2}$ and $\lambda_F = 10^{-2}$. Node i has 4 state regions determined by $TH_1 = 1.0$, $TH_2 = 2.0$, and $TH_3 = 3.0$. $S_1 = [0, 2.0]$, $S_2 = (2.0, \infty)$.

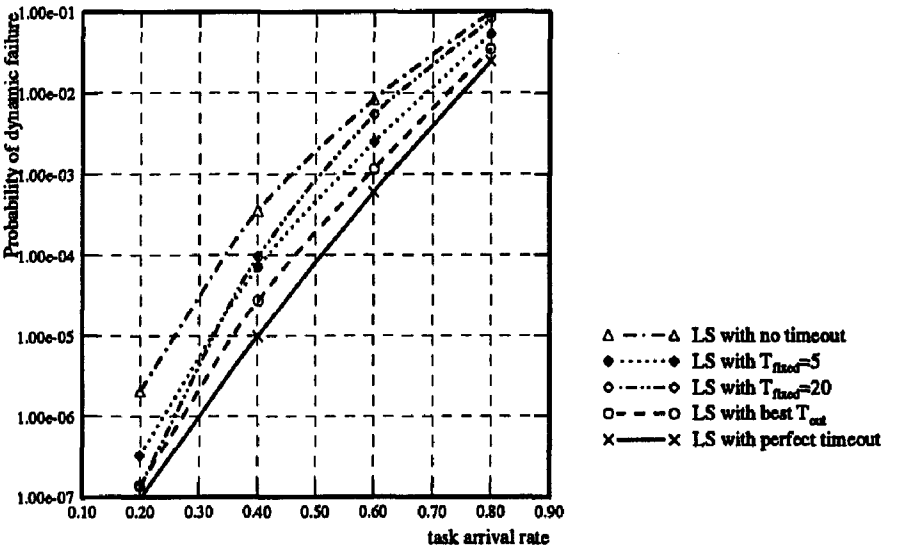


Figure 6.8: Performance comparison w.r.t. P_{dyn} for different timeout periods in a unreliable 16–node ($K_{np} = 16$) system. Local task attributes for node i: $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1.0, 2.0, 3.0\}_{\frac{1}{3}}$, and $K = 5$. $\lambda_F = 10^{-2}$, $\mu_F = 0.1$, and $\alpha_{ht} = 0.05$. The task-transfer delay is assumed to be 10% of the execution time of the transferred task. Each node has 4 state regions determined by $TH_1 = 1.0$, $TH_2 = 2.0$, and $TH_3 = 3.0$. $S_1 = [0, 2.0]$, $S_2 = (2.0, \infty)$.

inability of early detection of a node failure, thus increasing the possibility of sending overflow tasks to a failed node. The case with $T_{fixed}^{<i>} = 5$ is inferior to that with $T_{out}^{<i>}$ because of the undesirable effects of false diagnosis (i.e., deterioration of LS capacity). Frequent 'I am alive' messages in case of $T_{out}^{<i>}$ also consume communication bandwidth and compete with transferred tasks and/or regular broadcast messages for the use of communication medium when the system load ranges from medium to heavy. Thus, there is a definite performance advantage with on-line parameter estimation of task attributes and calculation of $T_{out}^{<i>}$. The performance with $T_{out}^{<i>}$ is, however, worse than the ideal case with immediate and perfect detection of node failures due to the fact that node n might keep sending its overflow task to a failed node i during the period between the occurrence of node i's failure and its detection by node n.

- A smaller $T_{fixed}^{<i>}$ is preferable as the system becomes more prone to node failures, especially for medium to heavy system loads (i.e., external task arrival rate ≥ 0.5). For example, the case of $T_{fixed} = 5$ outperforms the case of $T_{fixed} = 20$ in a more error–prone system (Fig. 6.8) as external task arrival rate ≥ 0.6. The performance improvement of frequent timeouts is, however, not as pronounced for a reliable system

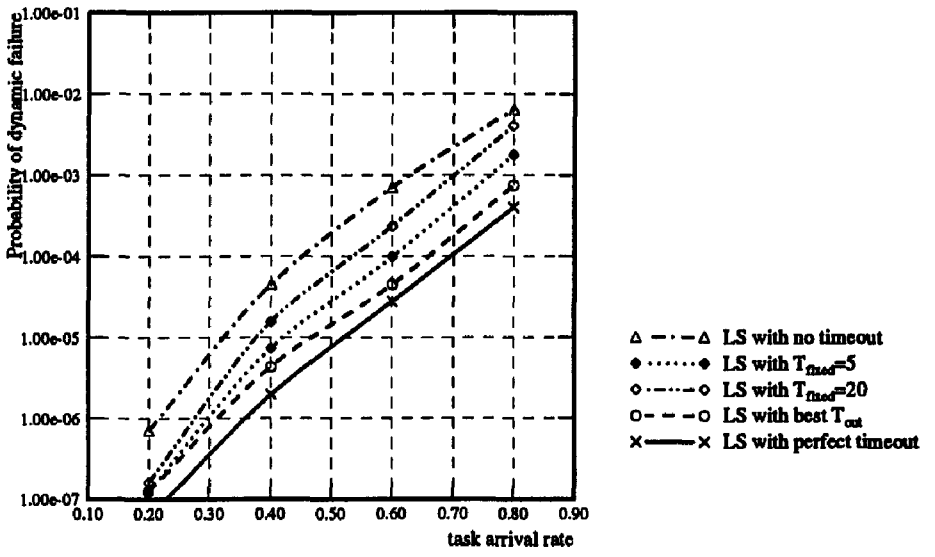


Figure 6.9: Performance comparison w.r.t. P_{dyn} for different timeout periods in a unreliable 64–node ($K_{np} = 64$) system. Local task attributes for node i: $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1.0, 2.0, 3.0\}_{\frac{1}{3}}$, and $K = 5$. $\lambda_F = 10^{-2}$, $\mu_F = 0.1$, and $\alpha_{ht} = 0.05$. The task–transfer delay is assumed to be 10% of the execution time of the transferred task. Each node has 4 state regions determined by $TH_1 = 1.0$, $TH_2 = 2.0$, and $TH_3 = 3.0$. $S_1 = [0, 2.0]$, $S_2 = (2.0, \infty)$.

(Fig. 6.8). This can be explained by the fact that as the nodes in a system become more prone to node failures, the performance deterioration caused by false diagnoses will get better compensated by the performance improvement due to early detection of node failures.

- As shown in Fig. 6.8 vs. Fig. 6.9, the effects of false diagnoses become less pronounced for the case with a smaller $T_{fixed}^{<i>}$ (e.g., $T_{fixed}^{<i>} = 5$) as the system size gets large. This is due to the fact that a larger system has a larger processing capacity and is thus more resilient to the deterioration of LS capacity caused by false diagnoses.

Impact of statistical fluctuation in task arrivals on the effectiveness of Bayesian estimation: We examined the effect of statistical fluctuation in task arrivals on the estimation of composite task arrival rates by simulating different task sets with hyperexponential external task inter-arrival times and varying the coefficient of variation (CV) of the inter–arrival time. Fig. 6.10 shows the simulation results under a heavy system load $\overline{\lambda_{ext}} = \lambda_i^{ext} = 0.8$ (where the performance is most sensitive to the variation of CV) with the window of the sample size $N_S = 30$. Also shown in Fig. 6.10 are the curves for the

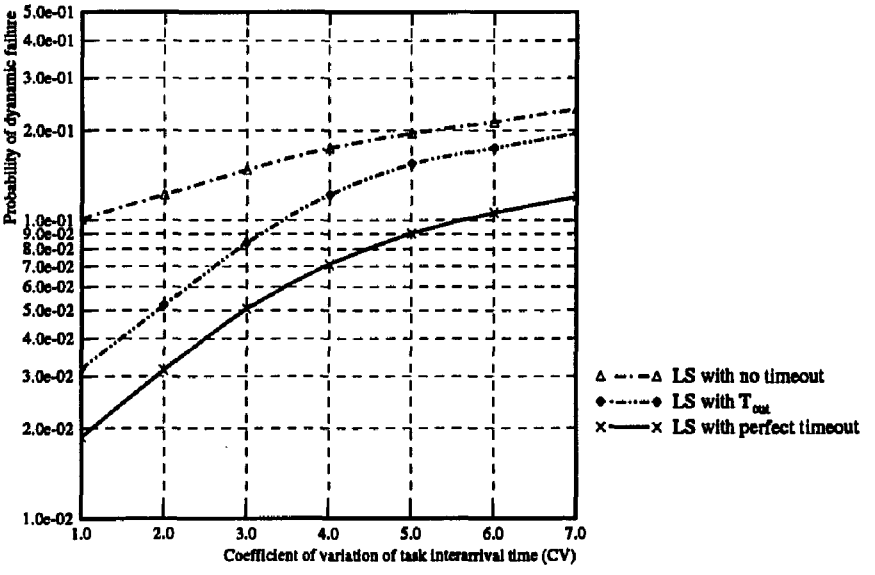


Figure 6.10: P_{dyn} vs. CV of external task interarrival times in a 16–node ($N = 16$) system. Local task attributes for node i: $ET = \{0.4, 0.8, 1.2, 1.6\}_{0.25}$, $L = \{1.0, 2.0, 3.0\}_{\frac{1}{3}}$, and $K = 5$. $\lambda_F = 10^{-2}$ and $\mu_F = 0.1$. The task–transfer delay is assumed to be 10% of the execution time of the transferred task. Each node has 4 state regions determined by $TH_1 = 1.0$, $TH_2 = 2.0$, and $TH_3 = 3.0$. $S_1 = [0, 2.0]$, $S_2 = (2.0, \infty)$.

case with no timeout mechanism and the case with immediate, perfect detection of node failures. As CV gets larger, the sample–mean based estimate deviates more from the true composite arrival rate λ_i due to the fact that the variability effect of task burstiness cannot be completely smoothed out. This accounts, in part, for the performance degradation of the proposed LS mechanism. However, the proposed timeout mechanism remains effective for all the task sets tested. For example, in Fig. 6.10, the performance of LS with $T_{out}^{<i>}$ is still 25% better than LS without any timeout mechanism. This suggests that within a wide range of task burstiness, the λ_i obtained from Bayesian estimation, although it might deviate from the true λ_i, is good for the calculation of $T_{out}^{<i>}$.

6.6 Conclusion

We proposed a timeout mechanism which, when there are node failures, can be incorporated into LS with aperiodic state–change broadcasts. By (1) on–line collection/estimation of parameters relevant to task attributes, and (2) calculating — based on the observation and the estimated task attributes in the latest broadcast — the best timeout period used to diagnose a silent node as failed, the probability of dynamic failure can be significantly reduced, as compared to LS without any timeout mechanism or with a fixed

timeout mechanism.

The negative impact of statistical fluctuation in task arrivals on the proposed timeout mechanism (in particular, Bayesian estimation) is also shown to be tolerable within a wide range of task arrival burstiness; for example, the performance of LS with $T_{out}^{<i>}$ is still 25% better than LS without a timeout mechanism.

Optimizing the tradeoffs involved with the timeout mechanism is an interesting design problem of its own. For example, there is a tradeoff between the potential performance improvement gained by reducing the broadcast-threshold interval and the performance deterioration resulting from the traffic overhead of region–change broadcasts. This kind of optimization is a matter of our future inquiry.

CHAPTER 7

AN EXAMPLE: INTEGRATED LS ON HARTS

7.1 Introduction

As discussed in Chapter 1, distributed systems are considered a suitable candidate for real-time applications due to their potential for high performance and high reliability. To realize part of this potential, at the Real-time Computing Laboratory of the University of Michigan we are currently building an experimental distributed real-time system, called HARTS (Hexagonal Architecture for Real-Time Systems) [Shi91].

In this chapter, we consider the issue of LS in HARTS, and adapt the concept of buddy sets, preferred lists, and region-change broadcasts proposed in Chapter 3 to HARTS [Shi91]. The nodes in HARTS are interconnected by the underlying interconnection topology — a C-wrapped hexagonal mesh[1] — and coordinated to evenly share "overflow" tasks. The HARTS routing and broadcasting algorithms in [CSK90, KS91b] are used for transferring tasks and broadcasting state-changes. The virtual cut-through switching scheme[2] [KK79] implemented in HARTS [DRS91] is used for inter-node communication. Moreover, by exploiting/integrating features of these algorithms for/into LS, we rigorously analyze the performance of LS in HARTS while considering all LS-related communication activities.

We first construct a continuous-time Markov chain to describe task arrival, transfer, and/or completion activities under the proposed LS mechanism. Second, we derive the traffic overheads introduced by LS (i.e., the rates of task transfer and state-change broadcast) from the Markov chain. Then, using the LS traffic rates as input and characterizing the hexagonal mesh topology and the virtual cut-through switching scheme implemented in HARTS, we construct a queueing network model from which the distribution of packet delivery time is derived. Finally, we derive the distribution of task waiting time (i.e., the time

[1] to be defined in Section 7.2.1.

[2] to be discussed in Section 7.2.1

a task is queued for execution plus the delivery time the task experiences if it is transferred), from which the probability of dynamic failure can be computed.

The impact of communication activities/delays on LS was first analyzed in [MTS89b, MTS89a]. They developed a continuous-time Markov chain for LS schemes with state probing, and characterized task transfer delays as with some percentage of mean task service time, but did not consider the underlying communication topology and the switching scheme used. We alleviated the negative effect of communication delays on maintaining up–to–date state information by using Bayesian analysis in Chapter 3 and in [SH91, HS91]. Although we considered the hypercube as the underlying interconnection system and included the effect of possible task/message queueing at intermediate nodes on the way to the destination node of each transfer/broadcast, the performance evaluation conducted in Chapters 3 and 4 does not consider the underlying communication subsystem along with its task routing, state-change broadcasting, and message-passing functions as an integrated part of the LS mechanism. To the best of our knowledge, this is the first to analytically evaluate the integrated LS performance using P_{dyn} as a yardstick.

The rest of this chapter is organized as follows. Section 7.2 gives a brief description of HARTS along with its routing and broadcasting algorithms, and its switching scheme. The LS mechanism proposed in Chapter 3 is then adapted to HARTS in Section 7.2. Section 7.3 deals with an integrated performance analysis of the proposed LS mechanism, the underlying communication subsystem, and the interaction between them. Section 7.4 describes a simulator for modeling the LS operations in HARTS (and for verifying our analysis), and presents some representative analytic/simulation results. Section 7.5 highlights the possible extension of our analysis to other LS schemes and/or interconnection structures, and concludes this chapter.

7.2 System Model and Load Sharing Mechanism for HARTS

7.2.1 Overview of HARTS

HARTS is an experimental distributed real-time system being built at the Real-Time Computing Laboratory, the University of Michigan [Shi91]. A set of Application Processors (APs) along with a Network Processor (NP) form a node of HARTS. These nodes are interconnected via a C-wrapped hexagonal mesh topology. The APs execute computational tasks, and the NP (which contains a custom-designed router, buffer memory, a RISC processor, and the interface to APs) handles both intra- and inter-node communications.

Specifically, a C-wrapped hexagonal mesh (H-mesh) can be defined succinctly as

follows.

Definition 1 *A C-wrapped hexagonal mesh of dimension e, denoted as H_e, is comprised of $M = 3e(e-1)+1$ nodes, labeled from 0 to $M-1$, such that each node k has six neighbors $[k+1]_M$, $[k+3e-1]_M$, $[k+3e-2]_M$, $[k+3e(e-1)]_M$, $[k+3e^2-6e+2]_M$, and $[k+3e^2-6e+3]_M$, where the dimension of an H-mesh is defined as the number of nodes on a peripheral edge of the H-mesh, and $[a]_b$ denotes a mod b.*

C-type wrapping is performed in the following three steps:

S1. Partition the nodes of a non-wrapped H-mesh of dimension e into rows in three different directions, d_0, d_1, d_2. The mesh can be viewed as composed of $2e-1$ horizontal rows (the d_0 direction), $2e-1$ rows in the 60-degree clockwise direction (the d_1 direction), or $2e-1$ rows in the 120-degree clockwise direction (the d_2 direction).

S2. Label from the top the rows L_0 through L_{2e-1} in each direction.

S3. Connect the last processor in L_i to the first processor in $L_{[i+e-1]_{2e-1}}$.

For example, Fig. 7.1 shows a simple C-wrapped H-mesh of dimension 5.

C-wrapped H-meshes have several nice properties as reported in [CSK90]. First, C-type wrapping results in a simple, transparent addressing scheme, where the center node is labeled as node 0, and the other nodes are labeled in sequence along the d_0 direction. (An example of the addressing for an H_5 is shown in Fig. 7.1.) Second, C-type wrapping results in a homogeneous network. Every node may view the mesh as a set of concentric hexagons (where each hexagon has one more node on each edge than the one immediately inside of it) with itself as the center node. Consequently, all nodes are topologically equivalent. Third, the diameter of an H_e is $e-1$. Consequently, any routing/broadcast packet traverses at most $e-2$ intermediate nodes before reaching its destination node. Fourth, simple and efficient routing and broadcast algorithms can be devised, as discussed in [CSK90, KS91b].

In what follows, we summarize the important features of the routing algorithm [CSK90], the broadcast algorithm [KS91b], and the virtual cut-through switching scheme [DRS91], all of which support LS-related communication activities. These will be exploited in our analysis of the proposed LS mechanism.

Features of Routing Algorithm: The routing algorithm derived from the simple addressing scheme determines all shortest paths between any two nodes given their addresses. In particular, it determines the number of hops, k_0, k_1, and k_2, from the source node to the destination node along the d_0, d_1, and d_2 directions, respectively. Note that all shortest

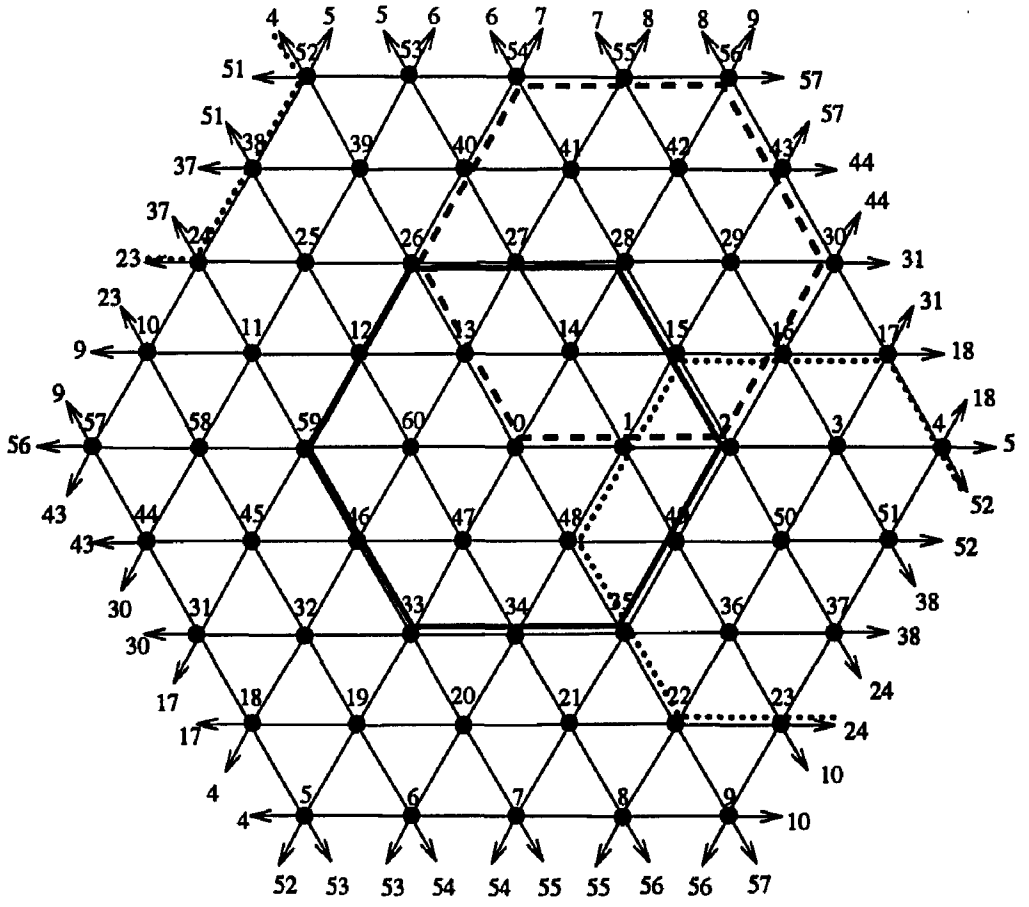


Figure 7.1: A C–wrapped hexagonal mesh of dimension 5, H_5.

paths are completely specified by k_i, $i = 0, 1, 2$, and the number of shortest paths with k_i hops in direction d_i is $(|k_0|+|k_1|+|k_2|)!/(|k_0|!\ |k_1|!\ |k_2|!)$. All shortest paths between any two nodes are assumed to be equally used with probability $(|k_0|!\ |k_1|!\ |k_2|!)/(|k_0| + |k_1| + |k_2|)!$. Another notable feature is that at each node on a shortest path there are at most two different neighbors of the node to which the shortest path runs, i.e., at most two of k_i, $i = 0, 1, 2$, are non-zero [CSK90].

Features of Broadcast Algorithm: An example of simple broadcasting in an H_4 is shown in Fig. 7.2, where, without loss of generality, the broadcasting node is placed at the center (node 0). For notational convenience, the $6h$ nodes which are h hops away from node 0 are said to be on the *h-th ring* centered at node 0, $1 \leq h \leq e - 1$. The algorithm propagates a packet, ring by ring, toward the periphery of the mesh. The broadcasting node generates and transmits six copies of each broadcast message, one along each direction, d_i,

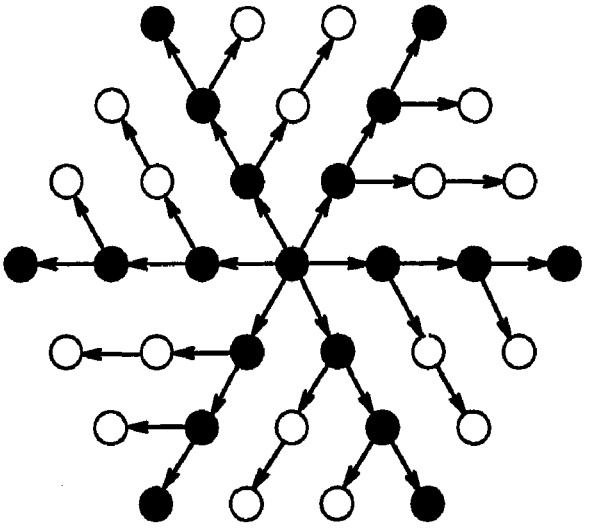


Figure 7.2: Simple broadcast for a hexagonal mesh of dimension 4, H_4. Corner nodes are shaded. Links between nodes are not drawn for clarity.

$0 \leq i \leq 5$. Upon receiving a broadcast message, if the node is a corner node[3] relative to the broadcasting node (node 0) — the node lies along the direction, d_i ($0 \leq i \leq 5$) with respect to node 0 — then it transmits the broadcast packet along the direction 60-degree clockwise to the direction from which the packet arrived, in addition to propagating the packet to the next node along the direction of packet receipt. Otherwise, it just forwards the packet to the next node along the same direction in which it is received.

Virtual Cut-Through: One of the switching schemes that support message routing and broadcasting in HARTS is commonly referred to as *virtual cut-through switching*. For this type of switching, packets arrived at an intermediate node are forwarded to the next node in the route without getting buffered if a circuit to the next node can be established. Specifically, the routing controller of HARTS contains six pairs of receivers and transmitters that are connected to a single *time-sliced*[4] bus. This bus is interfaced to the buffer management unit (BMU) in the node to store packets that cannot cut through the node, or are generated by or destined for the node. The operations for handling packets work as follows: when a packet is received, the receiver first recognizes the packet type. If the packet is of regular type, the receiver examines the routing tags in the packet header to check whether or not the packet has reached its destination. If not, it checks the directions in which a packet can be forwarded and tries to reserve a transmitter in one of these directions. If the reser-

[3]Note that there are 6 corner nodes on each ring.

[4]Each receiver is guaranteed to have access slot to the bus so that it may place on the bus the data it receives.

vation succeeds, the transmitter accepts the packet and forwards it to the next node (i.e., a cut–through occurs). If the reservation attempts do not succeed, the receiver requests BMU to store the packet for later transmission. If the received packet is of broadcast type, the receiver attempts to reserve the transmitter in the same direction, and drops a copy of the packet in the BMU simultaneously. Also, if the receiving node is a corner node, it will attempt to send a copy of the packet in the direction 60-degree clockwise to the direction from which the packet was received. If this reservation attempt does not succeed, the copy in the BMU will be transmitted later.

7.2.2 Adapting the Proposed Load Sharing Mechanism to HARTS

We now incorporate the LS mechanism proposed in Chapter 3 into HARTS. Salient features of this LS mechanism are that

1. It exploits the topological properties of the underlying interconnection network to evenly distribute overflow tasks over the entire system by using the concept of overlapping buddy sets and preferred lists in both location and information policies.
2. It reduces LS-related traffic while keeping the state information of other nodes as up–to–date as possible by using region-change broadcasts for the information policy.

The problem of systematically constructing buddy sets and preferred lists for hypercubes has been addressed in [SC89a]. We discuss here how to construct preferred lists in a C-wrapped H-mesh to minimize the probability of more than one node simultaneously transferring their overflow tasks to the same node.

Due to the homogeneity of a C-wrapped H-mesh, any node can consider itself as the center node (node 0) of the mesh. So, without loss of generality, we can concentrate on constructing the preferred list of node 0. Each node on the "ring" of h hops away from node 0 (or simply the h-th ring of node 0) can be reached from node 0 by a sequence of directions,

$$\underbrace{d_i\ d_i\ \ldots.\ d_i}_{j\text{ items}}\ \underbrace{d_{[i+1]_6}\ d_{[i+1]_6}\ \ldots\ d_{[i+1]_6}}_{(h-j)\text{ items}} \triangleq d_i^j\ d_{[i+1]_6}^{(h-j)},$$

for some i and j, where we have used the shortest–route feature of routing algorithm in Section 7.2.1. Also note that all permutations of $d_i^j\ d_{[i+1]_6}^{(h-j)}$ lead us to the same node, i.e., all sequences of directions which are composed of j d_i's and $(h-j)$ $d_{[i+1]_6}$'s lead us to the same node. Consequently, the address of each node can be uniquely determined by the sequence of directions as follows.

Lemma 1 *The node reachable from node i with any permutation of the sequence of directions, $a_1 a_2 ... a_k$, $a_j \in \{d_0, d_1, ..., d_5\}$, $\forall j \in \{1, \ldots, k\}$, has address $[i + \ell_0 + \ell_1(3e^2 - 6e + 3) + \ell_2(3e^2 - 6e + 2) + \ell_3(3e^2 - 3e) + \ell_4(3e^2 - 1) + \ell_5 3e^2]_{3e^2-3e+1}$, where ℓ_j $(0 \le j \le 5)$ is the number of times d_j appears in the sequence $a_1 a_2 ... a_k$, and e is the dimension of the H-mesh.*

The proof of this lemma follows directly from the recursive use of Definition 1. For example, the node reachable from node 0 with the sequence, $d_0\, d_1^2$, has address $[0 + 1 + 2(3 \cdot 5^2 - 6 \cdot 5 + 3)]_{3 \cdot 5^2 - 3 \cdot 5 + 1} = [97]_{61} = 36$ in an H_5 (Fig. 7.1).

Now, let the $6h$ nodes in the h-th ring be ordered as

$$\begin{array}{llllllll} d_0^h, & \overline{d}_0^h, & d_0^{(h-1)} d_1, & \overline{d}_0^{(h-1)} \overline{d}_1, & d_0^{(h-2)} d_1^2, & \overline{d}_0^{(h-2)} \overline{d}_1^2, & \ldots., & d_0 d_1^{(h-1)}, \quad \overline{d}_0 \overline{d}_1^{(h-1)}, \\ d_1^h, & \overline{d}_1^h, & d_1^{(h-1)} d_2, & \overline{d}_1^{(h-1)} \overline{d}_2, & d_1^{(h-2)} d_2^2, & \overline{d}_1^{(h-2)} \overline{d}_2^2, & \ldots, & d_1 d_2^{(h-1)}, \quad \overline{d}_1 \overline{d}_2^{(h-1)}, \\ d_2^h, & \overline{d}_2^h, & d_2^{(h-1)} d_3, & \overline{d}_2^{(h-1)} \overline{d}_3, & d_2^{(h-2)} d_3^2, & \overline{d}_2^{(h-2)} \overline{d}_3^2, & \ldots, & d_2 d_3^{(h-1)}, \quad \overline{d}_2 \overline{d}_3^{(h-1)}, \end{array} \tag{7.1}$$

where $\overline{d}_i \triangleq d_{[i+3]_6}$. Node 0's preference in transferring an overflow task is then determined ring by ring, beginning with the innermost ring and terminating at the outermost ring. (Nodes within each ring are ordered as above.) Specifically, the k-th preferred node of node 0 can be determined as follows: (1) find h such that $\sum_{j=1}^{h-1} 6j \le k < \sum_{j=1}^{h} 6j$, i.e., h determines which ring the k-th preferred node lies on; (2) set $\ell = k - \sum_{j=1}^{h-1} 6j$ specifies the position of the k-th preferred node within the h-th ring. The address of the k-th preferred node can then be determined by Eq. (7.1) and Lemma 1.

In what follows, we show that the preferred lists constructed above satisfy the requirements stated in Section 3.2.

Lemma 2 *Each node in an H-mesh will be selected as the k-th preferred node by one and only one other node, $1 \le k \le 3e(e-1)$.*

Proof: An H-mesh forms a homogeneous processing surface where a sequence of directions, $d_i^j\, d_{[i+1]_6}^{(h-j)}$, leading to a given destination uniquely determines the corresponding source node. Thus, the lemma follows from the way preferred lists are constructed (i.e., k uniquely determines h and ℓ which in turn uniquely determine $d_i^j\, d_{[i+1]_6}^{(h-j)}$). □

Lemma 3 *If node i is the k-th preferred node of node j, then node j is the $(k+1)$-th ($(k-1)$-th) preferred node of node i if k is odd (even).*

Proof: Suppose one follows the sequence, $d_\ell^m\, d_{[\ell+1]_6}^{(h-m)}$ to reach node j from node i, for some h, m, and ℓ. (Note that h, m, and ℓ are uniquely determined by k.) Then one can reach

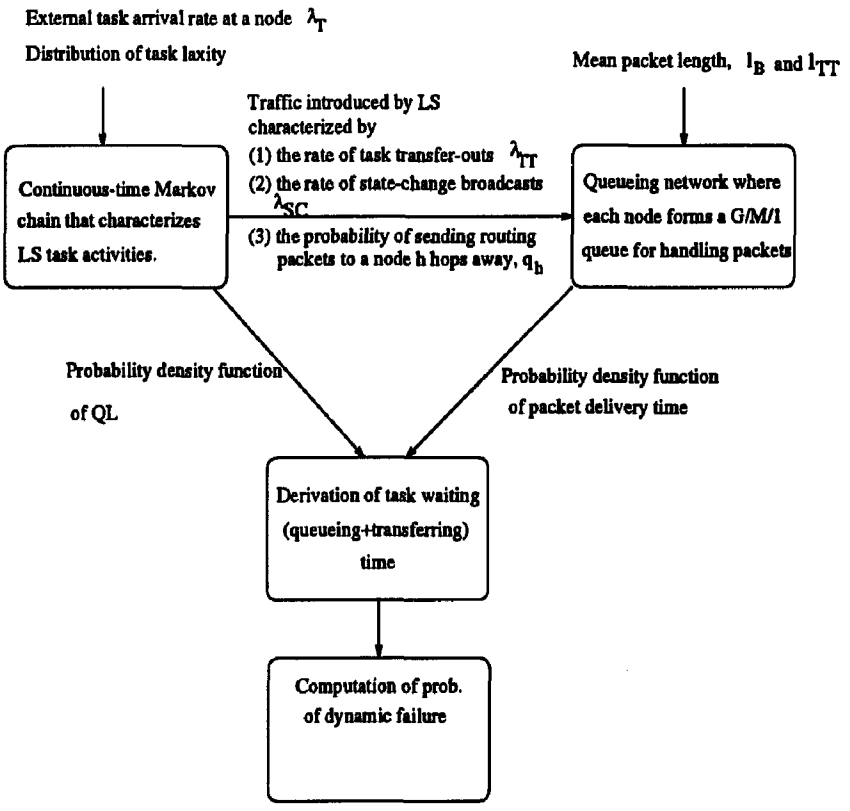


Figure 7.3: Analysis methodology used for evaluating the integrated LS performance. The continuous–time Markov chain and the queueing network can be accommodated for other LS schemes and interconnection structures as long as parameters are properly characterized.

node i from node j by following the sequence, $\overline{d}_\ell^m\ \overline{d}_{[\ell+1]_e}^{(h-m)}$, where $\overline{d}_\ell^m\ \overline{d}_{[\ell+1]_e}^{(h-m)}$ is the direction right after (before) $d_\ell^m\ d_{[\ell+1]_e}^{(h-m)}$ within the h-th ring in Eq. (7.1) if k is odd (even). □

This semi-symmetry property[5] implies that the task flow in one direction be approximately counter-balanced by that in the opposite direction. Now, the buddy set of a node can be formed with the first N_B nodes from the top of its preferred list. For ease of analysis, we assume that N_B is chosen such that the buddy set itself is also an H-mesh of dimension $m < e$, and $N_B = 3m(m-1)+1$.

7.3 Performance Analysis

Our analysis method is outlined in Fig. 7.3. We first construct an embedded continuous–time Markov chain to characterize task arrival/completion/transfer activities

[5]Unlike hypercubes, the symmetry property — if node i is the k-th preferred node of node j, then node j is the k-th preferred node of node i — cannot be achieved by any ordering of nodes due to the fact that the number of nodes in an H-mesh is odd.

under the proposed LS mechanism in HARTS. Two sets of parameters are derived from the constructed Markov model: (1) the probability density function of QL, $p_N(n)$, $n \geq 0$, and (2) the rate of transferring tasks, λ_{TT}, the rate of state-change broadcasts, λ_{SC}, and the probability of transferring an overflow task to a node h hops away, q_h. The latter set of parameters is fed into the queueing network where the handling of (both task–transfer and broadcasting) packets at each node in HARTS is modeled as a $G/M/1$ queue. The probability density function of packet delivery time, $f_{D_t}(t)$, can then be derived from the queueing network model and is used, along with $p_N(n)$, $n \geq 0$, to derive the probability density function of task waiting time, $f_{W_k}(t)$. Finally, the probability of dynamic failure, P_{dyn}, can be computed from $f_{W_k}(t)$.

7.3.1 LS Analytic Model

As discussed in Section 7.2.1, the nodes in HARTS are topologically homogeneous and are identical in processing capability and speed. Besides, all nodes are assumed to have the same arrival rate of 'external' tasks. Consequently, the task arrival/transfer activities experienced by each node are stochastically identical over a long term. Under this assumption, we employ the general methodology of first modeling the state evolution of a single node in isolation and then combining node-level models into a system-level model (discussed in Chapter 4).

We first model the state evolution of a node by a continuous-time Markov chain that serves as the underlying model. The parameters of this model are then derived to characterize task arrival/transfer/completion activities at the system level under the proposed LS strategy. Finally, a two-step iterative approach is taken to obtain a numerical solution to the Markov chain.

To facilitate the analysis, we make the following assumptions:

A1. External task arrivals at a node are Poisson with rate λ. Task execution times are exponentially distributed with mean $\frac{1}{\mu_T}$.

A2. Tasks are independent of one another, and are queued/executed on a node on a FCFS basis.

A3. Each task is associated with a laxity ℓ (in units of mean task execution time) with probability $\hat{p}_\ell$, $0 \leq \ell \leq L_{max}$, where L_{max} is the maximum task laxity.

A4. The composite (both external and transferred) task arrivals can be approximated as Poisson. Also implied in this assumption is that task arrival/departure activities on a node are approximately independent of those on others over a long term.

A1–A2 are consistent with those assumptions commonly used in the open literature [WM85, ELZ86, MTS89b, MTS89a, TT89]. The reason for **A2** is two-fold: first, FCFS is more analytically tractable than other local scheduling disciplines (e.g., MLFS discipline); second, as shown in Section 3.5 of Chapter 3, the choice of a local scheduling discipline has only a minimal effect on the *qualitative* assessment of LS performance. As was discussed in Chapters 4, **A4** serves only as an *approximation*; however, **A4** is known to become realistic as the system size grows, was used implicitly in [SC89a, MTS89b, MTS89a, TT89], and has been verified (via simulations) in Chapter 5 to be valid for systems of reasonably large size (≥ 12).

Underlying Model: The state, N, of a node is defined as its queue length (QL), and each node is modeled as an $M/M/1$ queue. The composite (both external and transferred) task arrival rate at a node with QL= n, denoted as $\lambda(n)$, depends on the node's QL, and the location and transfer policies used. The key issue in linking node-level models into a system-level model is to properly specify $\lambda(n)$, $n \geq 0$, so as to describe task arrivals and transfers in the system level. Once $\lambda(n)$, $n \geq 0$, is specified, the QL density function of a node, $\{p_N(n), n \geq 0\}$, can be obtained by solving

$$p_N(n) = p_N(0) \cdot \frac{\prod_{k=0}^{n-1} \lambda(k)}{\mu_T^n}, \quad n \geq 0, \quad \text{and} \quad \sum_{n=0}^{L_{max}+1} p_N(n) = 1. \tag{7.2}$$

Let $\alpha(n)$, $\beta(n)$, $\gamma_j = P(N \geq j)$ and K_j be defined similarly as in Section 4.3.2 of Chapter 4 except that now the state, N, is defined as the QL (instead of CET). Then, we have (see Fig. 4.3),

$$\lambda(n) = \lambda - \alpha(n) + \beta(n). \tag{7.3}$$

By appropriately specifying $\alpha(n)$ and $\beta(n)$ to describe both the transfer and location policies, one can use a embedded Markov chain model to describe the operations of the proposed LS mechanism. Following the same derivation in Section 4.3.2 of Chapter 4, we have

$$\alpha(n) = \lambda \cdot \sum_{j=0}^{n-1} \hat{p}_j, \tag{7.4}$$

$$\beta(n) = \sum_{j=n}^{L_{max}} \lambda \, \hat{p}_j \cdot \gamma_{j+1} \cdot \frac{1 - \gamma_{j+1}^{N_B}}{1 - \gamma_{j+1}}. \tag{7.5}$$

Two–Step Iterative Approach: $\lambda(n)$ and $\beta(n)$ must be known before solving the Markov chain model for $\{p_N(n), n \geq 0\}$ (see Eq. (7.2)). However, $\lambda(n)$ depends on γ_j which in turn depends on $p_N(n)$. Again an iterative approach is taken to handle the difficulty associated

with this recursiveness. In the first step, $p_N(n)$ is obtained by solving Eq. (7.2) with both $\alpha(n)$ and $\beta(n)$ set to 0, or, equivalently, $\lambda(n) = \lambda$, $\forall n$. The resulting $p_N(n)$'s are used to compute and $\beta(n)$ and $\lambda(n)$ in the second step. Then $p_N(n)$'s are re-calculated with the new $\beta(n)$ and $\lambda(n)$. This procedure repeats until both $p_N(n)$ and $\lambda(n)$ converge to some fixed values.

7.3.2 LS-Related Traffic: Derivation of λ_{TT}, λ_{SC}, and q_h:

In this subsection, we derive (1) the rate of transferring tasks out of a node, λ_{TT}, (2) the rate of state-change broadcasts, λ_{SC}, and (3) the probability of sending a task (in the form of a task–transfer message) to a node h hops away, q_h, to characterize LS-related communication activities. As shown in Fig. 7.3, these parameters model the interaction between LS and the underlying communication subsystem.

Derivation of λ_{TT}: Since a task with laxity j arrived at a node has to be transferred if QL $\geq j+1$, the task transfer-out rate, λ_{TT}, of a node can be expressed as

$$\lambda_{TT} = \sum_{j=0}^{L_{max}} \lambda \, \hat{p}_j \, \gamma_{j+1},$$

or,

$$\lambda_{TT} = \sum_{j=0}^{L_{max}} \lambda \hat{p}_j \cdot \sum_{k=j+1}^{\infty} p_N(k) = \sum_{k=1}^{L_{max}+1} p_N(k) \cdot \lambda \sum_{j=0}^{k-1} \hat{p}_j = \sum_{k=1}^{L_{max}+1} p_N(k) \cdot \alpha(k).$$

Derivation of λ_{SC}: Recall that there exist K_T state regions defined by $(K_T - 1)$ thresholds, TH_1, TH_2,..., $TH_{(K_T-1)}$ in the state (QL) space, and a node broadcasts a message, informing the other nodes in its buddy set of its state-region change whenever its state crosses any of the broadcast thresholds. Thus, the rate of state-change broadcasts, λ_{SC}, is related to the mean recurrence time, T_{ii}, of the i-th state region, R_i, $1 \leq i \leq K_T$. Specifically, T_{ii} is the expected time until the first transition into the i-th state region given that the node starts in the i-th state region, and λ_{SC} can be expressed as:

$$\lambda_{SC} = \sum_{i=1}^{K_T} \frac{1}{T_{ii}}.$$

T_{ii}, $1 \leq i \leq K_T$, can be derived from the continuous-time Markov chain, $\mathcal{M}$, constructed in Section 7.3.1. Specifically, by using the property of regenerative processes: for an irreducible, positive recurrent, and non-lattice continuous-time Markov chain (such as $\mathcal{M}$), we have

$$T_{ii} = \frac{T_i}{p_N(R_i)},$$

where T_i is the average time $\mathcal{M}$ spends in R_i, and can be expressed[6] as

$$T_1 = \frac{1}{\lambda(0)} \cdot \frac{\pi_0}{\sum_{k \in R_1} \pi_k} + \sum_{n \in R_1, n \geq 1} \frac{1}{\mu_T + \lambda(n)} \cdot \frac{\pi_n}{\sum_{k \in R_1} \pi_k},$$

and

$$T_i = \sum_{n \in R_i} \frac{1}{\mu_T + \lambda(n)} \cdot \frac{\pi_n}{\sum_{k \in R_i} \pi_k}, \quad i \geq 2,$$

where π_k is the stationary probability of the *underlying discrete-time* Markov chain for $\mathcal{M}$ stays in state k. $p_N(R_i)$ is the probability of $\mathcal{M}$ being in R_i, and can be expressed as

$$p_N(R_i) = \sum_{n \in R_i} p_N(n).$$

Derivation of q_h: Recall that node i transfers its overflow task to the first capable node found in its preferred list. By the way preferred lists are constructed, node i will transfer an overflow task to a node h hops away only if the first $3h(h-1)$ nodes (that are within $h-1$ hops) cannot complete the task in time and at least one of the $6h$ nodes on the h-th ring can. By **A4** and Lemma 3, q_h can be expressed in terms of γ_j and $\hat{p}_j$ as:

$$q_h = \frac{1}{6h} \sum_{j=0}^{L_{max}} \gamma_{j+1}^{3h(h-1)} (1 - \gamma_{j+1}^{6h}) \cdot \hat{p}_j.$$

7.3.3 HARTS Queueing Network Model

Packet delivery in the H-mesh is modeled as a queueing network, where each of $3e(e-1)+1$ nodes forms a $G/M/1$ queue. Broadcast and task-transfer packets are generated at a node when its state region changes and when the node cannot complete a newly-arrived task in time, respectively. These occurrences do not follow Poisson. Fortunately, characterizing packet arrival patterns *only* in terms of the rate of state-change broadcasts, λ_{SC}, and the rate of task transfers, λ_{TT}, while keeping the packet arrival process general suffices to derive the probability density function of packet delivery time. Both λ_{SC} and λ_{TT} depend on task arrival and departure activities in a node (and hence the LS mechanism used), and serve as the connection between the LS model and the HARTS queueing network.

A packet arrived at a node may go to one of its six immediate neighbors if it has not reached the destination node, or may otherwise exit from the system. The delay that a packet experiences at an intermediate node depends on whether or not it can cut through that node. If the packet establishes a circuit to a neighboring node, it will experience a

[6] since the distribution of the time until the next transition occurs given that $\mathcal{M}$ has just entered a state n is exponentially distributed with rate $\mu_T + \lambda(n)$.

negligible delay (so, we assume it to be 0). Otherwise, the packet requires "services", i.e., buffering and later transmission.

Thanks to the node homogeneity of HARTS, we can concentrate on evaluating the distribution of delivery time only for those packets generated at the center node, node 0. We will derive the following parameters in sequence.

- Both transit and non-transit loads handled by node 0. By a 'transit task-transfer packet,' we mean a task-transfer packet traversing a node that is not destined for the node. By a 'transit broadcast packet,' we mean a broadcast packet that should be forwarded to the next neighbor node(s) (because it has not reached a periphery node relative to the broadcasting node).
- The probability, p_f, that a packet arrived (and receiving services if necessary) at a node will be forwarded to one of its neighboring nodes.
- The throughput rate, Λ, at a node.
- The probability, p_c, of a packet cutting through an intermediate node.
- The distribution, $f_{D_i}(t)$, of the time needed for a packet to travel i hops.

As will be clearer later, the derivation of one parameter depends on the other parameters derived before it. To facilitate the analysis, we make the following assumptions:

B1. The probability of a node sending a packet to a node h hops away is q_h which was determined in Section 7.3.2.

B2. All shortest paths between a pair of nodes are equally used for task–transfer packets.

B3. The lengths of broadcast packets and task–transfer packets are exponentially distributed with mean $\overline{\ell_B}$ and $\overline{\ell_{TT}}$, respectively.

B4. The length of a packet is regenerated at each intermediate node of its route independently of its length at other intermediate nodes.

B2 can be justified as follows. The routing algorithm in [CSK90] gives *all* shortest paths between a pair of nodes by specifying k_0, k_1, and k_2. Upon receipt of a task-transfer packet, the routing algorithm (designed in [DRS91]) determines whether or not the packet has reached its destination node (i.e., $k_0 = k_1 = k_2 = 0$). If not, the packet is forwarded along one of the directions with nonzero k_i (and the corresponding k_i is updated) if cut-through can be established in that direction. Since the system is homogeneous, the probability of establishing a cut-through in any one of the forwarding directions is assumed to be equal, meaning that all shortest paths are equally used for task-transfer packets. **B2** along with the

topological properties of C-wrapped H-meshes will be used to determine the total number of shortest routes passing through node 0 for all pairs of communicating nodes. **B3**—**B4** coincide with Kermani and Kleinrock's assumptions in [KK79], and **B4** is commonly referred to as the independence assumption [KK79]. Although **B4** is unrealistic in practice, several empirical studies in [KK79, KN74] have shown that the mean packet delay times computed under this assumption closely match the actual mean message delivery times.

We will use the following notation to describe different packet arrival rates:

- λ_B (λ_{TT}): the rate of generating broadcast (task-transfer) packets at a node.
- $\lambda_{A,B}$ ($\lambda_{A,TT}$): the rate of terminal broadcast (task-transfer) packets arrived at a node.
- $\lambda_{T,B}$ ($\lambda_{T,TT}$): the rate of transit broadcast (task-transfer) packets arrived at a node.

Derivation of Packet Arrival Rates:

Lemma 4 *λ_B, $\lambda_{A,B}$, and $\lambda_{T,B}$ are given by*

$$\begin{aligned} \lambda_B &= 6(m-1)\lambda_{SC}, \\ \lambda_{A,B} &= 3m(m-1)\lambda_{SC}, \\ \lambda_{T,B} &= 3(m-1)(m-2)\lambda_{SC}, \end{aligned}$$

where λ_{SC} is the rate of state-region changes at node 0, and $m < e$ is the dimension of a buddy set, H_m.

Proof: As indicated in Fig. 7.2, (1) the broadcasting node generates 6 packets per broadcast, (2) every non-periphery corner node, upon receiving (and storing) a broadcast packet, transmits an additional copy along the direction 60-degree clockwise to the direction of packet receipt. By (1), the rate of generating broadcast packets for node 0 is $6\lambda_{SC}$. Also, by the homogeneity property of an H-mesh, node 0 acts as a non-periphery corner node relative to $6(m-2)$ nodes in its buddy set (see Fig. 7.1), and is responsible for generating (and forwarding) a new packet upon arrival of a packet broadcast by those $6(m-2)$ nodes. Thus, by (2), node 0 generates broadcast packets for $6(m-2)$ other nodes in its buddy set at the total rate of $6(m-2)\lambda_{SC}$. The expression of λ_B thus follows from (1) and (2).

Node 0 receives broadcast packets from the other $3m(m-1)$ nodes in its buddy set, and since all nodes are homogeneous each with a state-change broadcast rate λ_{SC}, the expression of $\lambda_{A,B}$ follows. On the other hand, node 0 lies on the periphery of the buddy set of other $6(m-1)$ nodes whose broadcast packets will not be forwarded by node 0. That is, the non-transit load (destined for node 0) is $6(m-1)\lambda_{SC}$. $\lambda_{T,B}$ is thus $[3m(m-1) - 6(m-1)]\lambda_{SC} = 3(m-1)(m-2)\lambda_{SC}$. □

Lemma 5 *$\lambda_{A,TT}$ and $\lambda_{T,TT}$ are given by*

$$\lambda_{A,TT} = \lambda_{TT} \sum_{k=1}^{m-1} 6k^2 \cdot q_k,$$
$$\lambda_{T,TT} = \lambda_{TT} \sum_{k=2}^{m-1} 6k(k-1) \cdot q_k,$$

where λ_{TT} is the rate of transferring tasks out of node 0, and q_k is the probability of a node transferring tasks to a node k hops away.

Proof: To calculate $\lambda_{A,TT}$ at node 0, we need to determine (1) the load contributed to $\lambda_{A,TT}$ at node 0 by a shortest route that passes through node 0; (2) the total number of shortest routes passing through node 0 for all pairs of communicating nodes.

It was verified in [CSK90] that all shortest routes between any pair of nodes are formed by links along at most two directions and thus can be represented by a sequence of directions, $d_j^i\, d_{[j+1]_6}^{(\ell-i)}$, where $0 \le j \le 5$, ℓ is the length of the shortest route ($1 \le \ell \le m-1$), and i and $\ell - i$ ($1 \le i \le \ell$) are the number of hops from the source to the destination along d_j and $d_{[j+1]_6}$, respectively.

Now, consider a sequence of directions, $d_j^i\, d_{[j+1]_6}^{(\ell-i)}$. The number of shortest routes associated with $d_j^i\, d_{[j+1]_6}^{(\ell-i)}$ that pass through node 0 can be calculated as follows. There are $\binom{\ell}{i}$ permutations for d_j's and $d_{[j+1]_6}$'s in the sequence $d_j^i\, d_{[j+1]_6}^{(\ell-i)}$, each of which gives a possible shortest route. For each possible route, node 0 can be inserted in one of the ℓ positions (including the destination) to be an intermediate or destination node of the route. Thus, each sequence of the form $d_j^i\, d_{[j+1]_6}^{(\ell-i)}$ represents $\binom{\ell}{i} \cdot \ell$ shortest routes that pass through node 0.

The load at node 0 contributed by a single shortest route (represented by $d_j^i\, d_{[j+1]_6}^{(\ell-i)}$) that passes through node 0 can be expressed as $\lambda_{TT} \cdot q_\ell / \binom{\ell}{i}$, because a node sends packets to nodes ℓ hops away at a rate of $\lambda_{TT} \cdot q_\ell$ which is equally shared by all $\binom{\ell}{i}$ shortest routes (under **B2**).

The rate of task-transfer packets arrived at node 0 can now be expressed as

$$\lambda_{A,TT} = \sum_{j=0}^{5} \sum_{\ell=1}^{m-1} \sum_{i=1}^{\ell} \text{load contributed by all routes represented by } d_j^i\, d_{[j+1]_6}^{(\ell-i)}$$
$$= \sum_{j=0}^{5} \sum_{\ell=1}^{m-1} \sum_{i=1}^{\ell} \binom{\ell}{i} \cdot \ell \cdot \frac{\lambda_{TT} \cdot q_\ell}{\binom{\ell}{i}} = \sum_{j=0}^{5} \sum_{\ell=1}^{m-1} \sum_{i=1}^{\ell} \lambda_{TT} \cdot q_\ell \cdot \ell = \lambda_{TT} \sum_{\ell=1}^{m-1} 6\ell^2 \cdot q_\ell.$$

The proof of the second part of the lemma resembles the first part except that for each possible route that can be represented by $d_j^i\, d_{[j+1]_6}^{(\ell-i)}$, node 0 cannot be the destination

node. Thus, there are $\ell - 1$ positions to insert node 0, and each sequence $d_j^i\, d_{[j+1]_6}^{(\ell-i)}$ gives $\binom{\ell}{i} \cdot (\ell - 1)$ shortest routes that pass through node 0. The rate of transit traffic, $\lambda_{T,TT}$, at node 0 can now be expressed as

$$\begin{aligned}\lambda_{T,TT} &= \sum_{j=0}^{5} \sum_{\ell=1}^{m-1} \sum_{i=1}^{\ell} \text{transit load contributed by all routes represented by } d_j^i\, d_{[j+1]_6}^{(\ell-i)} \\ &= \sum_{j=0}^{5} \sum_{\ell=1}^{m-1} \sum_{i=1}^{\ell} \binom{\ell}{i} \cdot (\ell - 1) \cdot \frac{\lambda_{TT} \cdot q_\ell}{\binom{\ell}{i}} = \lambda_{TT} \sum_{\ell=1}^{m-1} 6\ell(\ell - 1) \cdot q_\ell. \qquad \square\end{aligned}$$

Derivation of p_f: The probability, p_f, that a packet arrived at a node will be forwarded to one of its neighboring nodes is the ratio of the traffic bound for immediate neighbors to all traffic arrived at a node:

$$p_f = \frac{\lambda_B + \lambda_{T,B} + \lambda_{TT} + \lambda_{T,TT}}{\lambda_B + \lambda_{A,B} + \lambda_{TT} + \lambda_{A,TT}}.$$

Using the results of Lemmas 4 and 5 along with

$$\sum_{k=1}^{m-1} 6k \cdot q_k = 1,$$

we have

$$p_f = \frac{3m(m-1) \cdot \lambda_{SC} + \sum_{k=1}^{m-1} 6k^2 q_k \cdot \lambda_{TT}}{3(m+2)(m-1) \cdot \lambda_{SC} + [1 + \sum_{k=1}^{m-1} 6k^2 q_k] \cdot \lambda_{TT}}. \tag{7.6}$$

Derivation of Throughput of a Node: To derive the throughput, Λ_i, of a node i, we use the principle of *flow conservation*.[7] Specifically, let Λ_i and Λ_k, $0 \le k \le 5$, represent the throughput of node i and its neighboring nodes in directions d_0–d_5, respectively, and let p_{f,d_k} represent the probability that a packet completing its service will be forwarded to the neighboring node in direction d_k, then the flow conservation principle enforces

$$\Lambda_i = (\lambda_B + \lambda_{TT}) + \sum_{k=0}^{5} p_{f,\overline{d_k}} \cdot \Lambda_k.$$

Now, by the homogeneity of the C-wrapped H-mesh, all Λ_i's are equal. Also, $\sum_{k=0}^{5} p_{f,\overline{d_k}} = p_f$. Thus, we have

$$\Lambda_i = \frac{\lambda_B + \lambda_{TT}}{1 - p_f} = \frac{3(m+2)(m-1) \cdot \lambda_{SC} + [1 + 6\sum_{k=1}^{m-1} k^2 q_k] \cdot \lambda_{TT}}{6(m-1)\lambda_{SC} + \lambda_{TT}} \cdot (6(m-1)\lambda_{SC} + \lambda_{TT}). \tag{7.7}$$

[7] which states that at any branching point in a queueing network there is no accumulation or loss of customers.

Derivation of p_c: As discussed in Section 7.2.1, a packet can cut through an intermediate node only if no packets are being transmitted or waiting at that node. Using the *Utilization Law* [Kle75] [8] the probability of having no packet at a node is $1-\rho$, where ρ is the traffic intensity

$$\rho = \frac{\Lambda_i}{\mu_m} = \frac{\Lambda_i \cdot \overline{\ell}}{6}.$$

Here $\overline{\ell}$ is the mean packet length, and can be expressed as

$$\overline{\ell} = \frac{\lambda_B + \lambda_{A,B}}{\lambda_{TT} + \lambda_{A,TT} + \lambda_B + \lambda_{A,B}} \cdot \overline{\ell_B} + \frac{\lambda_{TT} + \lambda_{A,TT}}{\lambda_{TT} + \lambda_{A,TT} + \lambda_B + \lambda_{A,B}} \cdot \overline{\ell_{TT}}.$$

p_c can then be expressed as

$$p_c = 1 - \frac{\Lambda_i \cdot \overline{\ell}}{6},$$

where Λ_i is expressed in Eq. (7.7).

Derivation of Distribution of Packet Delivery Time: The delivery time for a packet traveling i hops, denoted by D_i, depends on whether or not the packet can cut through intermediate nodes. If the packet cuts through an intermediate node (the probability of which is p_c), its delay at the node is negligible and assumed to be 0. Otherwise, the packet experiences an exponential buffering delay, Y_k, with rate $\mu_m(1-\rho)$.[9] Moreover, the probability of a packet not cutting through j out of $i-1$ intermediate nodes (when it travels i hops) is

$$\binom{i-1}{j} (1-p_c)^j \, p_c^{i-1-j}.$$

The distribution of D_i can then be expressed as

$$\begin{aligned} P(D_i \le t) &= \sum_{j=0}^{i-1} P(D_i \le t \mid \text{buffered at } j \text{ intermediate nodes}) \cdot P(\text{buffered at } j \text{ intermediate nodes}) \\ &= \sum_{j=0}^{i-1} P\Big(Y_0 + \sum_{k=1}^{j} Y_k + Y_i \le t\Big) \cdot \binom{i-1}{j} (1-p_c)^j \, p_c^{i-1-j}, \end{aligned}$$

where a packet always gets buffered at the source and destination nodes, and $Y_0 + \sum_{k=1}^{j} Y_k + Y_i$ can be shown to have an Erlang distribution with parameters $\mu_m(1-\rho)$ and $j+2$ under **B4**. That is, the probability density function of D_i can be expressed as

$$f_{D_i}(t) = \sum_{j=0}^{i-1} [\mu_m(1-\rho)]^{j+2} \cdot \frac{t^{j+1} e^{-\mu_m(1-\rho)t}}{(j+1)!} \cdot \binom{i-1}{j} (1-p_c)^j p_c^{i-1-j}.$$

---

[8] Note that the Utilization Law is valid for a G/G/1 queue.

[9] Note that the time experienced by a packet buffered at an intermediate node depends on the throughput of the node, and is accounted for by the factor $(1-\rho)$.

7.3.4 Derivation of Task Waiting Time and Probability of Dynamic Failure

Having derived the probability density function of QL, $\{p_N(n), n \geq 0\}$, and the probability density function of packet delivery time, $f_{D_i}(t), t \geq 0$, $1 \leq i \leq m-1$, we are now in a position to derive the distribution of task waiting time.

The probability density function of waiting time for a task queued on a node with state $N \leq k$ is

$$f_{W|N\leq k}(t \mid N \leq k) = \sum_{n=0}^{k} f_{W_{(n)}}(t \mid N = n) \cdot \frac{p_N(n)}{1-\gamma_{k+1}},$$

where $f_{W_{(n)}}(t \mid N = n)$ is the probability density function of waiting time given $N = n$, and can be shown under **A1** and **A2** in Section 7.3.1 to be n-stage Erlang. Thus,

$$f_{W|N\leq k}(t \mid N \leq k) = \delta(t) \cdot \frac{p_N(0)}{1-\gamma_{k+1}} + \sum_{n=1}^{k} \frac{\mu_T(\mu_T t)^{n-1}}{(n-1)!} e^{-\mu_T t} \cdot \frac{p_N(n)}{1-\gamma_{k+1}}, \qquad (7.8)$$

where $\delta(t)$ is the impulse function such that $\delta(t) = 0$ for $t \neq 0$; $\delta(t) \neq 0$ for $t = 0$, and $\int_0^\infty \delta(t)\, dt = 1$.

The waiting time for a task with laxity k depends on whether or not the task arrives at a node with QL $\leq k$. If the QL of the node at which the task arrives is $\leq k$, then the task experiences the waiting time with density function $f_{W|N\leq k}(t \mid N \leq k)$. Otherwise, the task has to be transferred, possibly several times until it arrives at a capable node. That is, the task experiences the delivery time(s) and the waiting time on a capable node. The possibility of multiple transfers results from the fact that the state at the selected receiver node i hops away at the time *when the task is sent by the sender node* may be different from that *when the transferred task arrives at the receiver node.* The possibility of state inconsistency increases as the packet delivery time, D_i, increases.

Specifically, let $f_{W|N>k}(t)$ denote the density function of the waiting time experienced by a task with laxity k that arrived at an incapable node and is thus transferred out, then the density function of waiting time for tasks with laxity k can be expressed as:

$$f_{W_k}(t) = (1-\gamma_{k+1}) \cdot f_{W|N\leq k}(t \mid N \leq k) + \gamma_{k+1} \cdot f_{W|N>k}(t), \qquad (7.9)$$

where $f_{W|N>k}(t)$ is approximated as:

$$f_{W|N>k}(t) = \sum_{i=1}^{m-1} \gamma_{k+1}^{3i(i-1)} (1-\gamma_{k+1}^{6i}) \cdot f_{D_i}(t) * [\, p_{sc,i}\, f_{W|N\leq k}(t \mid N \leq k) + (1-p_{sc,i})\, f_{W|N>k}(t)\,]. \qquad (7.10)$$

Here $*$ denotes the convolution of two probability density functions, and $p_{sc,i}$ is the probability of *state consistency* within an i-hop delivery time, which is derived below. Note that

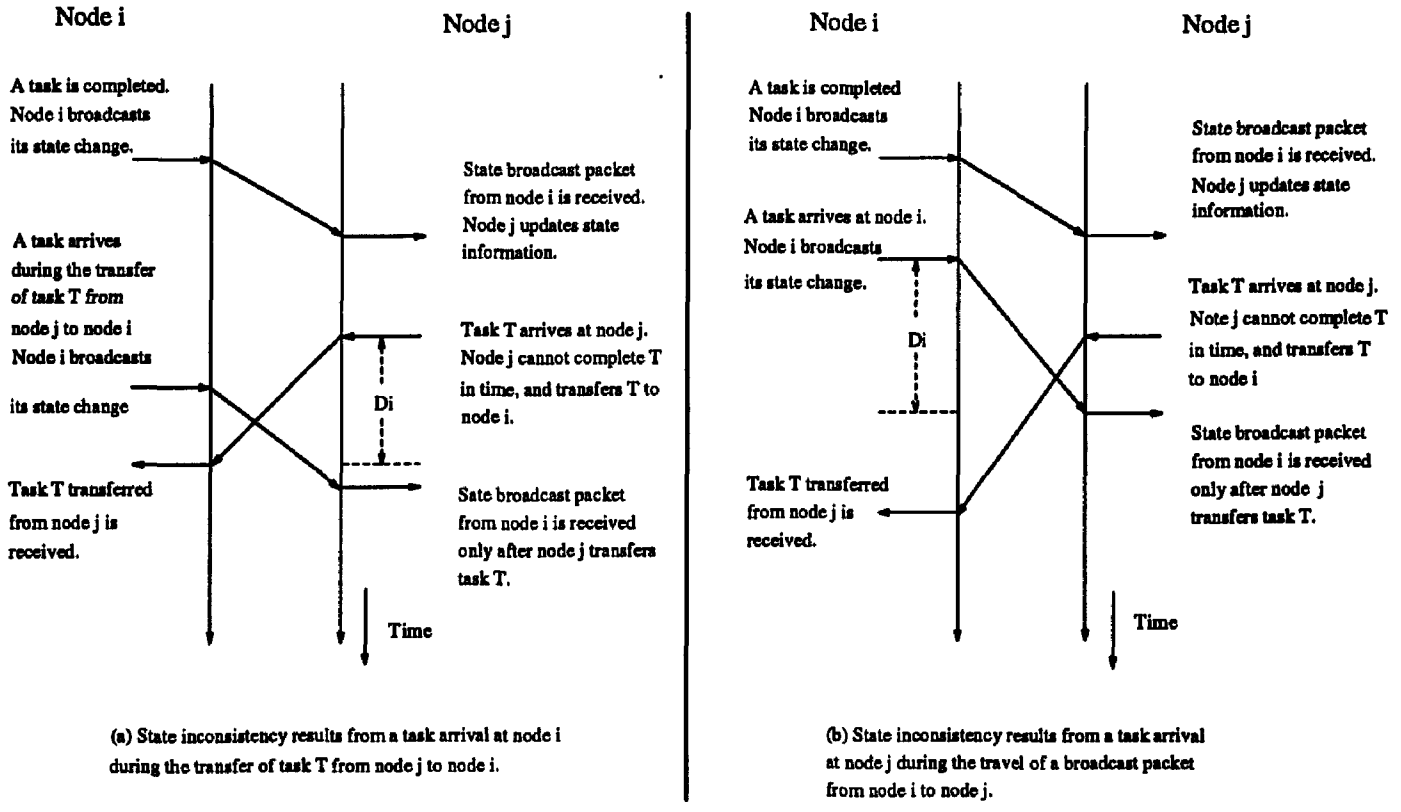


Figure 7.4: Two situations state inconsistency may arise.

$f_{W|N>k}(t)$ is expressed as a function of itself, thus a time–frequency domain transformation (e.g., the Laplace transform) is necessary to obtain numerical solutions for $f_{W|N>k}(t)$, $t \geq 0$.

To derive $p_{sc,i}$, two scenarios in which state inconsistency may occur are considered. In the first case (Fig. 7.4 (a)), during the transfer of an overflow task $\mathcal{T}$ from a sender node j to a receiver node i, a new task arrives at node i, making it become unable to complete the transferred task $\mathcal{T}$ upon its arrival. In the second case (Fig. 7.4 (b)), node j receives an overflow task $\mathcal{T}$ and transfers it to an incapable node i before the broadcast packet (informing node j of node i's incapability) from node i reaches node j. Note that in both cases state inconsistency arises because a task arrives during the packet delivery time, D_i. We thus approximate[10] $p_{sc,i}$ as the probability that no tasks arrive during the packet delivery time, D_i, i.e.,

$$\begin{aligned} p_{sc,i} &= P(\text{No task arrives within an } i\text{-hop delivery time}) \\ &= \int_0^\infty \{ \sum_{n=0}^{L_{max}+1} e^{-\lambda(n)t} \cdot p_N(n) \} \cdot f_{D_i}(t)\, dt. \end{aligned}$$

[10]Note that $p_{sc,i}$ errs on the conservative side, because the selected receiver node (node i in Fig. 7.4) of a transferred task $\mathcal{T}$ may still complete $\mathcal{T}$ in time after receiving a new task as long as the QL of the receiver node after receiving the new task is still less than or equal to the laxity of $\mathcal{T}$.

Finally, the probability of dynamic failure, $P_{dyn|k}$, experienced by a task with laxity k is

$$P_{dyn|k} = \int_k^{\infty} f_{W_k}(t)\, dt, \quad \text{and} \quad P_{dyn} = \sum_{k=0}^{L_{max}} \hat{p}_k \cdot P_{dyn|k}.$$

7.4 Numerical Examples

To evaluate the performance of LS while considering all related communication activities, we used a discrete-event simulator which models the operations of the proposed LS mechanism in HARTS. The routing, broadcasting, and virtual cut-through schemes in [CSK90, KS91b, DRS91] are used to facilitate LS-related communication activities. The goal of the simulation is two-fold: (1) examine the impact of approximations/assumptions made in the analysis, and (2) evaluate the integrated LS performance.

The simulator was originally developed by the authors of [DRS91] which accurately models the delivery of each packet by emulating the routing hardware along the route of a packet at the microcode level. It also captures the internal bus access overheads experienced by packets as they pass through an intermediate node. For example, when a packet arrives at a node, the following sequence of events is initiated. First, the receiver for the link on which the packet arrived waits for the packet header to become available. It then examines the packet header to determine the packet type. For a broadcast packet, the receiver tries to schedule two events: one to reserve the transmitter(s) to forward the packet(s), and the other to the BMU to receive the packet. For a task-transfer packet, it either may have arrived at its final destination in which case an event is scheduled on the BMU, or could be transmitted in one of possibly two directions. In the latter case, the receiver attempts to reserve the first-choice direction to transmit the packet. If the attempt to reserve the transmitter was unsuccessful, an attempt to try for an alternate transmitter is made, if applicable. If none of these attempts are successful, the packet is queued at this node for later transmission. Lastly, the receiver schedules events to signal the completion of the packet transmission at this node. This may involve un-reserving a transmitter if the packet successfully cuts through and/or informing the module which handles buffered messages.

We modified the simulator to include modules that (1) model task arrival, transfer, and completion activities under the proposed LS mechanism, (2) generate task-transfer and broadcast packets (along with their proper headers) at the time of task transfer and state-region change, and (3) update the preferred list of a node upon receipt of a broadcast packet, so that the main features of the proposed LS mechanism may be incorporated into the simulator. The simulator differs from the analytical model in that: (**D1**) if a node

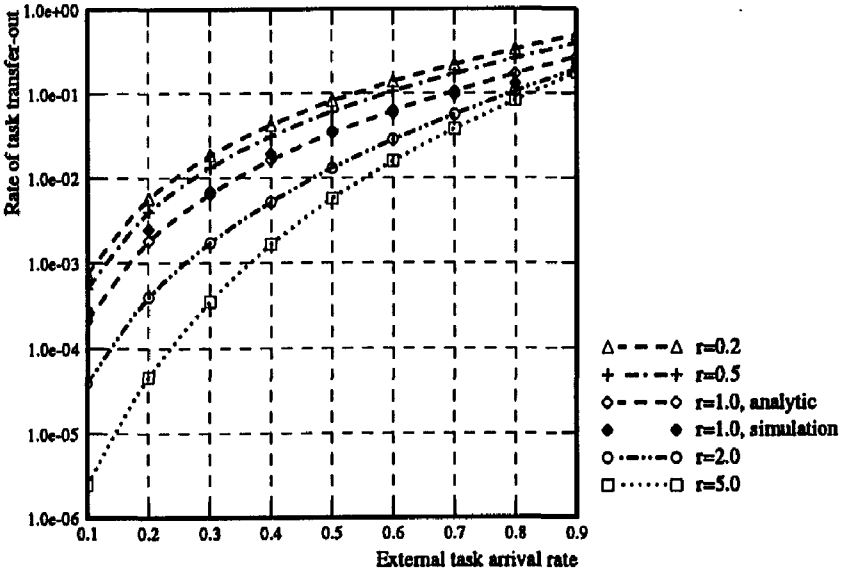
considers none of the nodes in its preferred list is capable of completing its overflow task in time, the overflow task is declared failed and taken into account of the statistics for P_{dyn}; **(D2)** the length of a packet is determined at the time of its birth and remains unchanged while the packet traverses through the network. The latter is used to inspect the discrepancy between the packet delivery time analytically computed under **A4** in Section 7.3.3 and that observed in practice.

An H_5 is used as an example for all simulation runs. The buddy set is chosen to be an H-mesh of dimension 4. For convenience, μ_T is set to 1, and all time-related parameters are expressed in units of $1/\mu_T$. The threshold values are set to $TH_1 = 1$, $TH_2 = 3$, and $TH_3 = 5$ unless specified otherwise. Simulations are carried out for a task set with the external task arrival rate λ on each node varying from 0.1 to 0.9, the mean task-transfer packet length $\overline{\ell_R}$ varying from 0.1 to 5.0, and the mean broadcast packet length $\overline{\ell_B}$ varying from 0.01 to 0.15. The distribution of task laxity is assumed to be a geometric distribution with $\hat{p}_{\ell+1} = r_{atio} \cdot \hat{p}_\ell$, where $1 \leq \ell \leq 5$, and r_{atio} is chosen as 0.2, 0.5, 1.0, 2.0, and 5.0. Note that $r_{atio} = 1.0$ gives a uniform distribution.

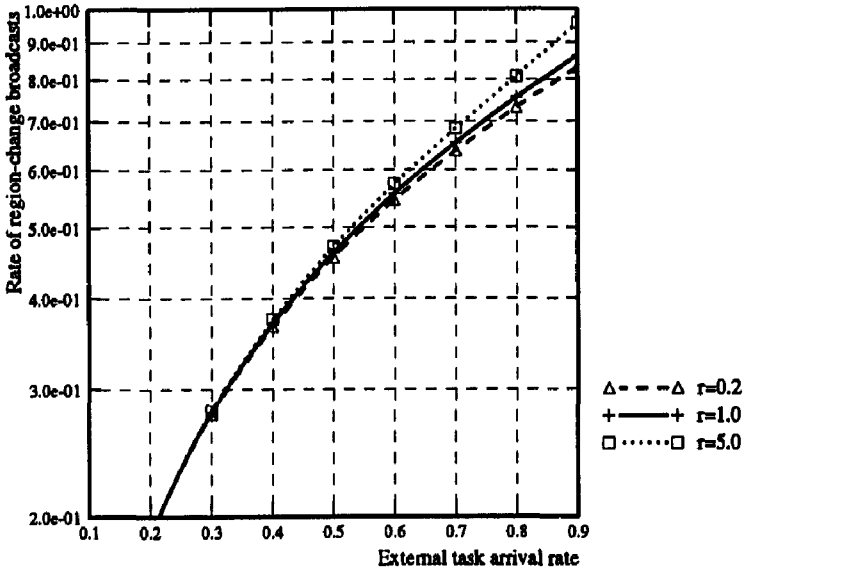
For each combination of parameters, the number of simulation runs needed is determined such that a 95% confidence level in the results for a maximum error of 5% of the specified probability can be achieved. We also compare the numerical results obtained with two other baselines whenever appropriate. The first baseline is an $M/M/1$ queue, representing the case of no load sharing, and the second baseline is an $M/M/37$ queue, representing the case of perfect load sharing where each node has perfect state information of other nodes in the buddy set and incurs no time overheads in task transfers and state-change broadcasts.

The traffic overhead introduced by LS and its impact on the possibility of cut-through are plotted in Fig. 7.5, where λ_{TT} vs. λ, λ_{SC} vs. λ, q_2 vs. λ, and p_{ct} vs. λ are plotted. As expected, λ_{TT}, λ_{SC}, and q_h all increase as λ increases. Consequently, cut-through is more unlikely to be established at intermediate nodes as λ increases (Fig. 7.5 (d)). λ_{TT} and q_h also increase as the task laxity gets tighter, but λ_{SC} decreases as the laxity gets tighter, where the tightness of laxity is measured in terms of r_{atio} as defined above. The latter phenomenon is perhaps due to the fact that an incapable node tends to locate *idle* nodes for its overflow tasks with tight laxities. Under such a scenario, both the sender node and the idle destination node need not broadcast a region-change message upon arrival of a tight-laxity task,[11] and hence λ_{SC} slightly decreases as task laxity gets tighter.

[11] Note that $TH_1 = 1$, and thus when QL changes from 0 to 1, no message is broadcast.

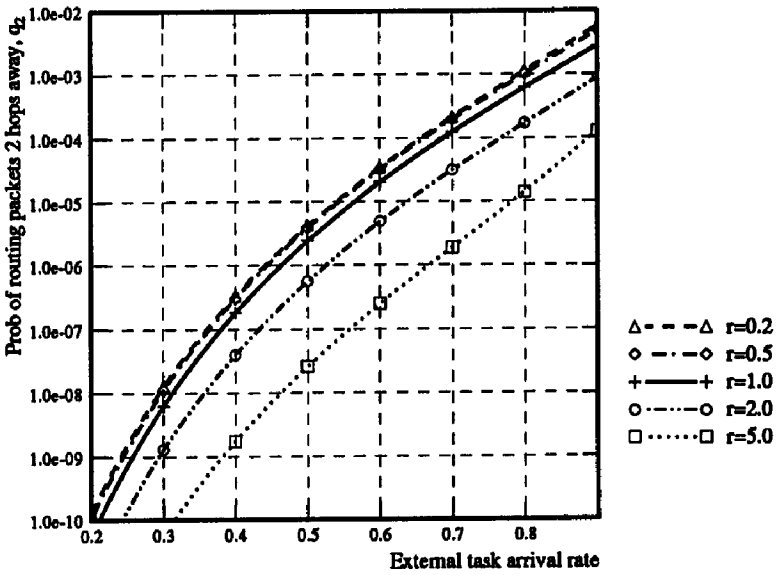


(a) λ_{TT} vs. λ.

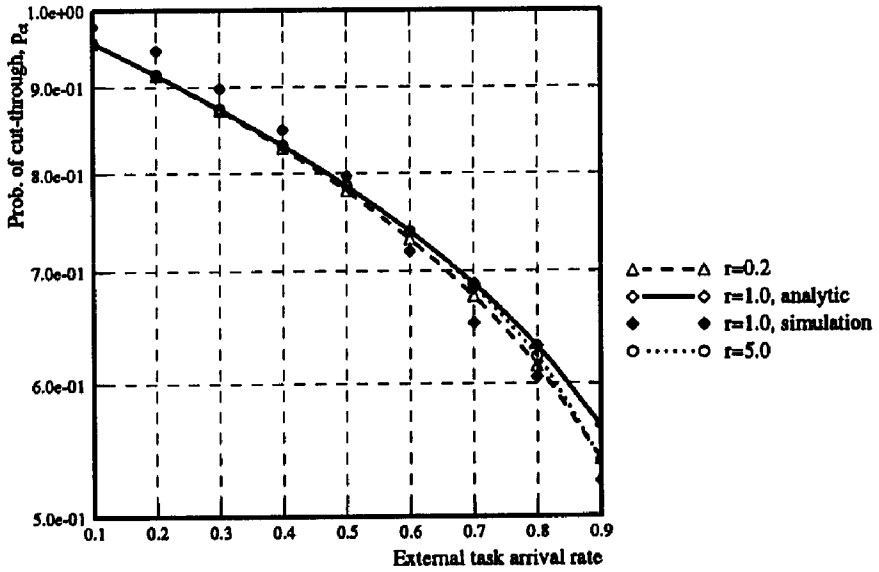


(b) λ_{SC} vs. λ.

Figure 7.5: Traffic generated by LS (measured in terms of λ_{TT},λ_B, q_h, and p_{ct}) for different external task arrival rate, λ. The distribution of task laxity is assumed to be uniformly distributed over [1,5]. $\overline{\ell_R} = 0.5$, $\overline{\ell_B} = 0.05$.



(c) q_2 vs. λ.



(d) p_{ct} vs. λ.

Figure 7.5: (continued) Traffic generated by LS (measured in terms of λ_{TT},λ_B, q_h, and p_{ct}) for λ.

As shown in Fig. 7.5(a) and (d), the analytical results predict, with a reasonable accuracy the simulation results (usually within a 6% difference in all our simulation studies). The fact that the analytic model overestimates p_{ct} at higher loads is perhaps because the model does not consider the overhead of processing packet headers. This overhead becomes non–negligible when the number of packets traversing the network is high.

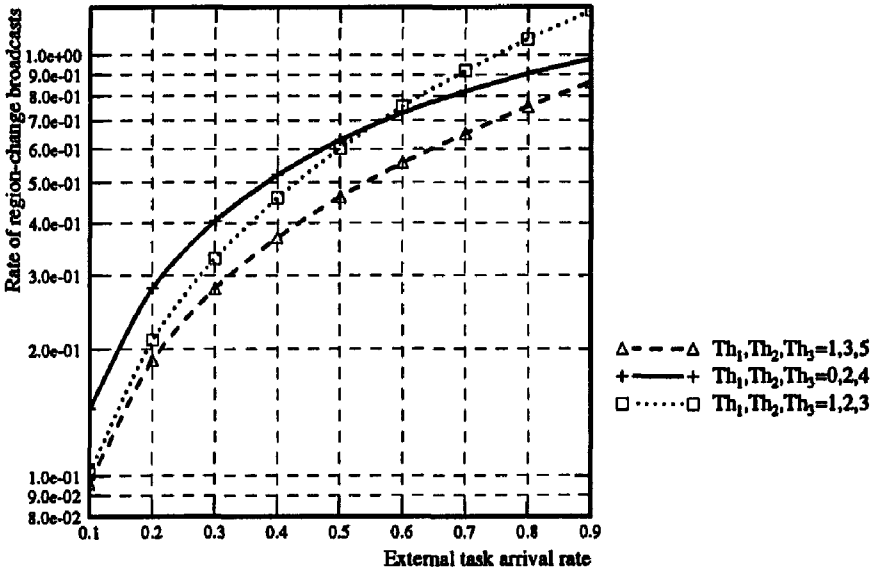
Fig. 7.6 shows the effect of threshold values on the traffic (λ_{SC} and p_{ct}) introduced by LS. Since a node broadcasts its state only when its QL crosses some threshold, threshold values are crucial to λ_{SC} (and thus p_{ct}). As suggested in Fig. 7.6, the threshold pattern of $TH_1 = 1, TH_2 = 3$, and $TH_3 = 5$ introduces less broadcast traffic as compared to the other patterns. The pattern of $TH_1 = 0, TH_2 = 2, TH_3 = 4$ performs worst at light system loads (because an idle node crosses $TH_1 = 0$ upon a task arrival), while the pattern of $TH_1 = 1$, $TH_2 = 2$, $TH_3 = 3$ performs worst at high task loads (because the arrival of every task causes a non-idle node's QL to cross thresholds).

Fig. 7.7 shows the plots of $1 - P_{dyn}$ vs. λ and $1 - P_{dyn}$ vs. r_{atio}. The proposed LS mechanism significantly outperforms the case of no LS (the $M/M/1$ system) especially at high system loads or tight task laxity, but is still inferior to perfect LS (the $M/M/37$ system). The latter also suggests that the time overheads in task transfers and state-change broadcasts deteriorate the LS performance and thus an efficient communication system that supports time-constrained communication is essential to real-time LS. The fact that the analytic model slightly overestimates $1 - P_{dyn}$ at higher loads is partly because of **D1** stated above.

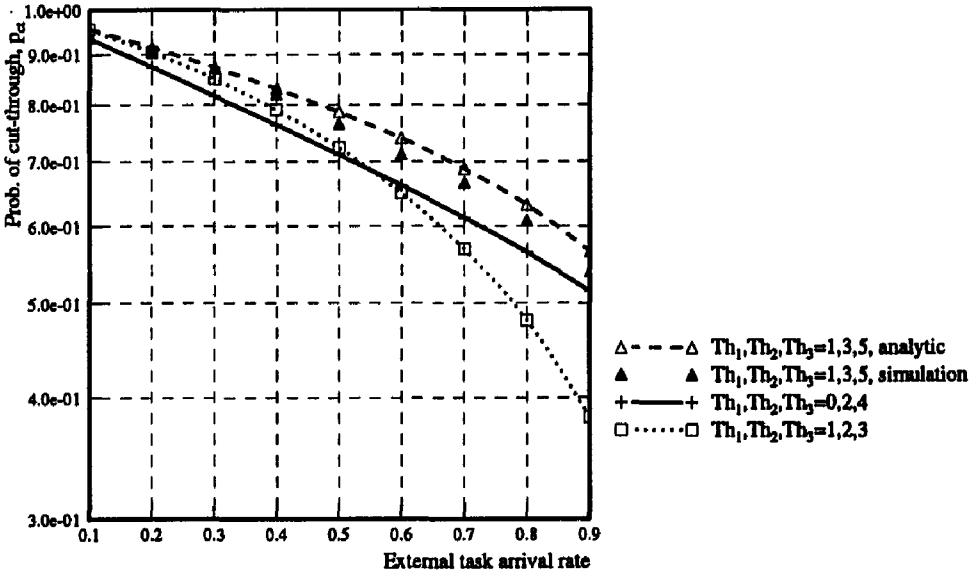
The impact of communication delays on the performance of LS is studied by varying both $\overline{\ell_R}$ and $\overline{\ell_B}$ (and consequently μ_m). Fig. 7.8 (a) and (b) show the plots of p_{ct} vs. $\overline{\ell_B}$ and $1 - P_{dyn}$ vs. $\overline{\ell_B}$, respectively. (The effect of varying $\overline{\ell_R}$ on LS is similar to, but less pronounced than,[12] that of $\overline{\ell_B}$, and thus omitted.) As shown in Fig. 7.8(a), p_{ct} drops abruptly as $\overline{\ell_B}$ increases beyond a certain value. For example, when $\lambda = 0.8$, p_{ct} equals 8.533×10^{-1}, 5.918×10^{-1}, 2.616×10^{-1}, and 2.089×10^{-2} at $\overline{\ell_B} = 0.01, 0.05, 0.1$, and 0.14, respectively. This indicates that when $\overline{\ell_B}$ becomes very large (or equivalently, the processing speed of the BMU is slow), the network becomes saturated and incapable of handling all incoming packets (introduced by LS). Consequently, packets queue up at every intermediate node, thus delaying or even blocking the operation of task transfers and state–change broadcasts, and hence $1 - P_{dyn}$ decreases until it reaches approximately the value of P_{dyn} of an $M/M/1$ system (no LS).

---

[12] This is because $\lambda_B = 6(n-1)\lambda_{SC}$ and $\lambda_{A,B} = 3n(n-1)\lambda_{SC}$ are usually much larger than λ_{TT} and $\lambda_{A,R}$, and thus dominate the determination of μ_m.

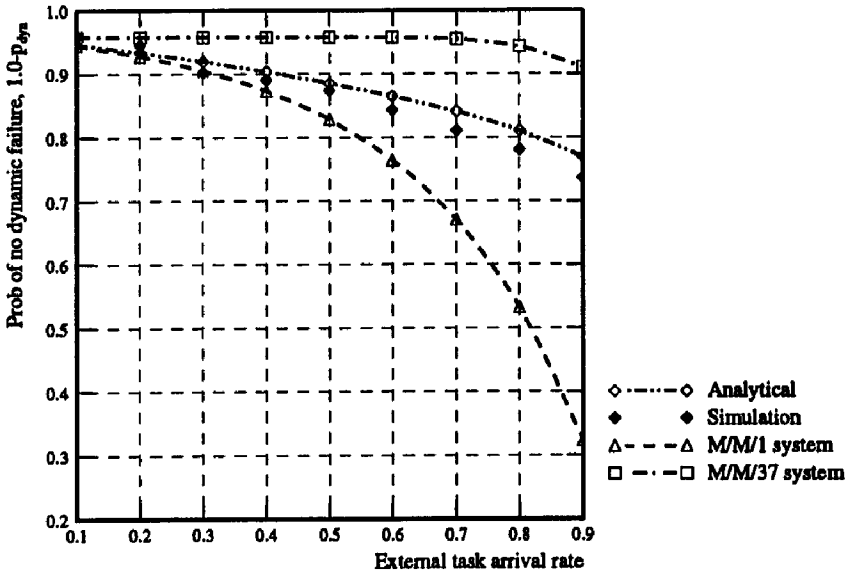


(a) λ_{SC} for different sets of threshold values.

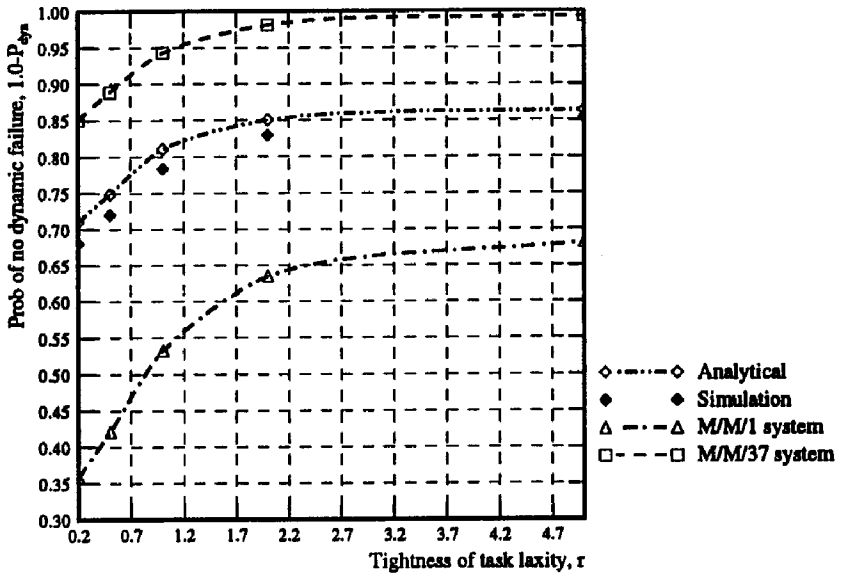


(b) p_{ct} for different sets of threshold values.

Figure 7.6: Effect of threshold values on the traffic generated by LS. The distribution of task laxity is assumed to be uniformly distributed over [1,5]. $\overline{\ell_R} = 0.5$, $\overline{\ell_B} = 0.05$.

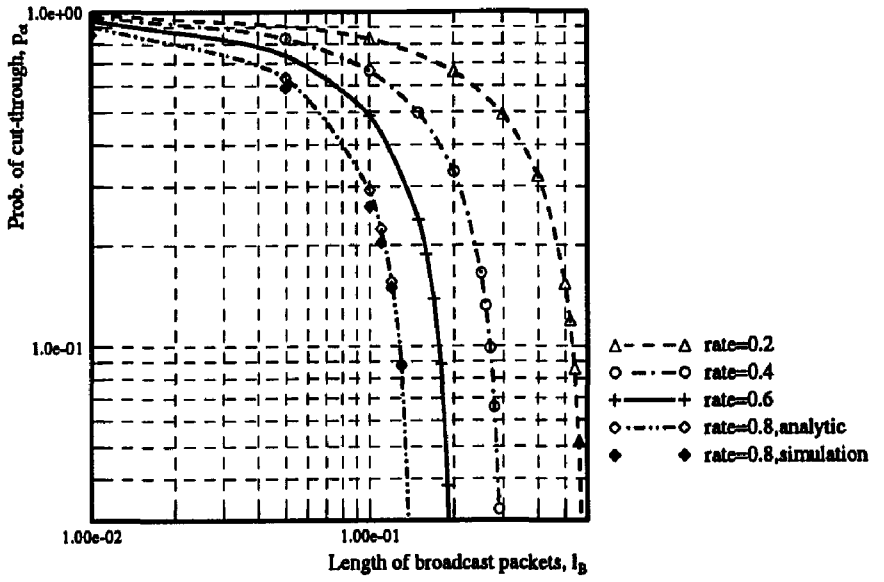


(a) $1 - P_{dyn}$ vs. λ.

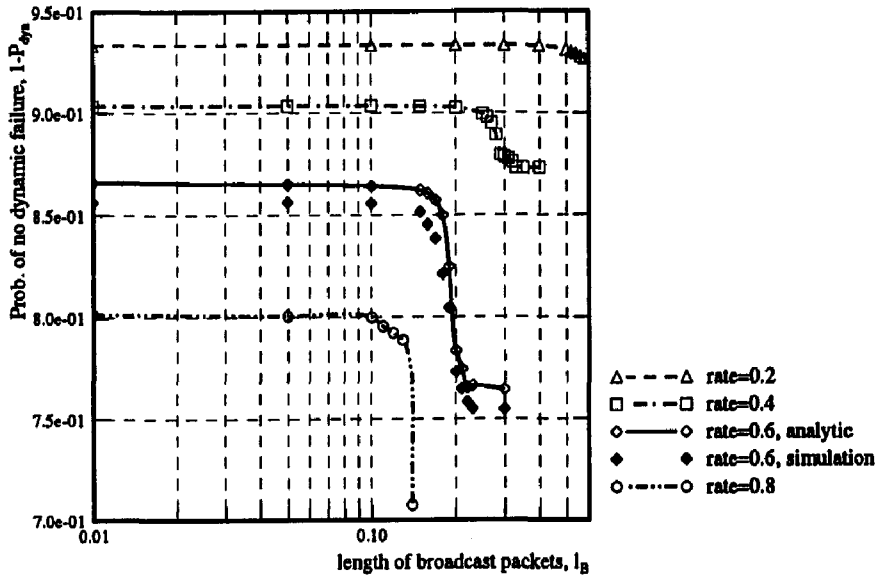


(b) $1 - P_{dyn}$ vs. r_{atio} ($\lambda = 0.8$).

Figure 7.7: $1 - P_{dyn}$ vs. λ and tightness of task laxity r_{atio}. The distribution of task laxity is uniformly distributed over [1,5] in (a), and is geometrically distributed with $\hat{p}_{\ell+1} = r \cdot \hat{p}_{\ell}$, for $1 \leq \ell \leq 5$. $\lambda = 0.8$, $\overline{\ell_R} = 0.2$, and $\overline{\ell_B} = 0.02$. $\overline{\ell_R} = 0.5$, $\overline{\ell_B} = 0.05$.



(a) p_{ct} vs. $\overline{\ell_B}$.



(b) $1 - P_{dyn}$ vs. $\overline{\ell_B}$.

Figure 7.8: The Effect of $\overline{\ell_B}$ on p_{ct} and $1 - P_{dyn}$. The distribution of task laxity is uniformly distributed over [1,5], and $\overline{\ell_R} = 0.5$.

7.5 Conclusion

The analysis presented in this chapter is essential to the design of any LS mechanism for real-time applications. First of all, the model gives a quantitative measure of traffic overheads (through λ_{TT}, λ_B, and $f_{D_i}(t)$) introduced by the LS mechanism. Second, one can determine many LS design parameters (e.g., the size of buddy sets, N_B, and the threshold values, TH_i's, for state-change broadcasts) by determining the parameter values which minimize P_{dyn}. Third, one can investigate whether or not the underlying routing and broadcasting schemes can support the LS mechanism in delivering packets and collecting/maintaining up–to–date state information.

The analysis methodology presented here (Fig. 7.3) is quite general in the sense that it can be extended to other LS mechanisms and other interconnection topologies such as hypercubes, rings, or square meshes, to name a few. Extending this methodology to other LS mechanisms, one only needs to properly derive the parameters $\alpha(n)$ and $\beta(n)$ which characterize the transfer policy and the location policy, respectively. Once $\alpha(n)$ and $\beta(n)$ are determined, the derivation of the others (e.g., $p_N(n)$, λ_{TT}, λ_B, and q_h) follows the same approach. Similarly, extending the methodology to other interconnection topologies, one only needs to specify the parameters p_f and p_c. The key to the specification of p_f (and consequently p_c) is the determination of transit loads at each node. That is, one must determine the fraction of shortest routes between all pairs of communicating nodes that pass through a given node as an intermediate node. Once p_f and p_c are determined, the derivation of packet delivery time distribution does not depend on the interconnection topology.

CHAPTER 8

IMPLEMENTATION BASED ON CONDOR

8.1 Introduction

As was discussed in Chapter 1, a LS mechanism can be designed by developing the transfer, information, and location policies. On the other hand, the implementation issues commonly considered include where to place the LS mechanism (inside or outside the OS kernel), how to transfer process state (virtual memory, open files, process control blocks) during task transfer/migration, and how to support LS transparency and reduce the effects of residual dependency[1] [DO91]. There have been a few LS mechanisms implemented, such as the V–system [TLC85], the Sprite OS [DO91], the Charlotte OS [AF89], and the Condor software package [LLM88, LL90]. They are designed with different policies for transferring tasks/processes, collecting workload statistics used for LS decisions, and locating target workstations. They are implemented with different strategies to detach a migrant process from its source environment, transfer it with its context (the per-process data structures held in the kernel), and attach it to a new environment on the destination workstation.

In this chapter, we describe a preliminary implementation of the decentralized LS mechanism proposed in Chapter 3 that is based on Condor. As reported in [LLM88, LL90], Condor is a software package for executing long–running tasks on workstations which would otherwise be idle. It is designed for a workstation environment in which the workstation's resources are guaranteed to be available to the owner of the workstation. The reason for choosing Condor as our "base system" is because Condor is implemented entirely outside the OS kernel and at the user level. This eliminates the need to access/change the internals of OS. On the other hand, there are several design drawbacks of Condor: it uses a central manager workstation to allocate queued tasks to idle workstations; that is, the location policy is entirely realized by the central manager. This centralized component makes the LS

[1] residual dependency is defined as the need for the source workstation to maintain data structures or provide functionality for a remote process.

mechanism susceptible to single–workstation failures. Another drawback is that Condor uses a periodic information policy; that is, each workstation reports periodically to the central manager regarding its (workload) state and task–queueing situation. This makes the central manager a potential bottleneck of network traffic from time to time. The determination of a reporting period also becomes crucial to the LS performance, and has to be traded off between the communication overhead introduced by frequent reporting and the possibility of using out–of–date state information resulting from infrequent reporting.

Hence, we enhance the fault-tolerance capability and the performance of Condor by configuring and dispatching the functions of the central manager workstation to multiple workstations, and "transforming" Condor to a decentralized LS mechanism. In particular, we use the preferred lists and region–change broadcasts proposed in Chapter 3 to achieve this goal. That is, each workstation collects/maintains elaborate and timely state information on its own by using region–change broadcasts. Moreover, each workstation determines, by using the preferred lists, the best target workstation for each task if there are several workstations available to reduce the possibility of multiple workstations sending their tasks to the same idle workstation.

The rest of the chapter is organized as follows. In Section 8.2, we give an overview of Condor software package and discuss how Condor daemons collaborate to manage the task queue and locates idle target workstations. In Section 8.3, we discuss how to get rid of the central manager by incorporating our decentralized LS mechanism and reconfiguring Condor component daemons. In Section 8.4, we highlight the implementation features adopted in the decentralized mechanism. In Section 8.5, we discuss related work and present alternative design and implementation approaches adopted by other existing process migration mechanisms. This chapter concludes with Section 8.6.

8.2 Overview of Condor Software Package

In this section, we summarize the functionality of, and the interactions among, Condor's daemons. Especially, the task distribution process is described in a step–by–step manner.

As shown in Fig. 8.1, there are two daemons, Negotiator and Collector, running on the central manager workstation. In addition, there are two other daemons, Schedd and Startd, running on each participating workstation. Whenever a task is executed, two additional processes, Shadow and Starter, shall run on the submitting workstation and on the executing workstation, respectively (whether or not these two workstations are actually

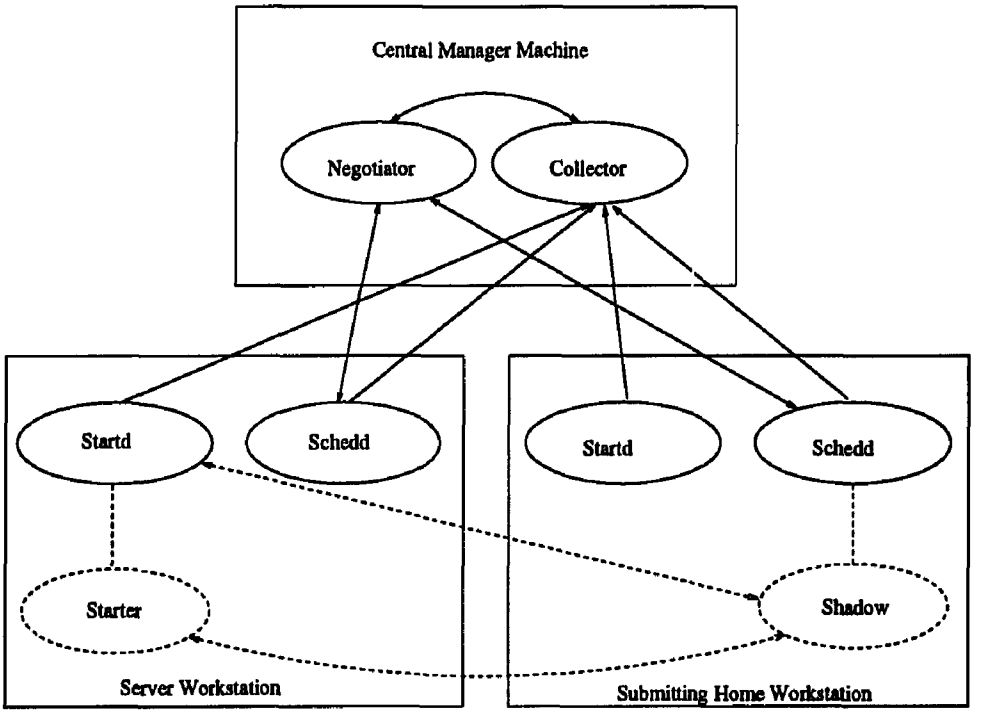


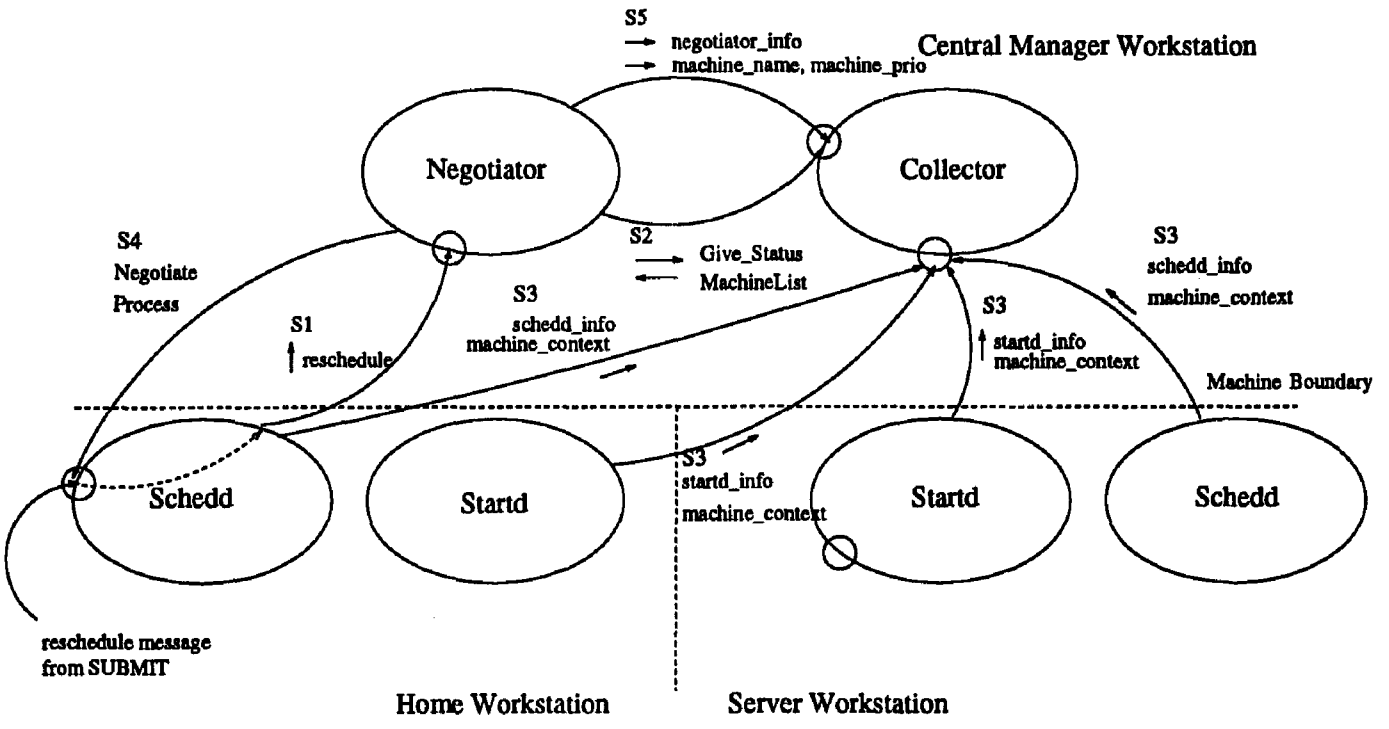
Figure 8.1: Daemons in Condor.

identical).

The Condor task relocation mechanism works as follows (Fig. 8.2). A user invokes a **submit** program to submit a task. The **submit** program takes the task description file, constructs the corresponding data structures, and sends a **reschedule** message to Schedd on the home workstation. Schedd then asks Negotiator on the central manager workstation to relocate tasks to idle workstations by sending a **reschedule** message to Negotiator (**S1** in Fig. 8.2).

Upon receiving a **reschedule** message from any of Schedds on the participating workstations, or upon periodic schedule timeout, Negotiator gets from Collector a list of machine records which contains the workload and task queue of all participating workstations (**S2** in Fig. 8.2). Collector receives periodically from Schedd and Startd on each participating workstation updated information of task queue and workload, respectively (**S3** in Fig. 8.2), and updates accordingly its list of machine records.

After receiving the list of machine records, Negotiator first prioritizes the participating workstations: the priority of a workstation is incremented by the number of individual users with tasks queued on that workstation, and decremented by the number of tasks which are submitted to that workstation and are currently running (either remotely



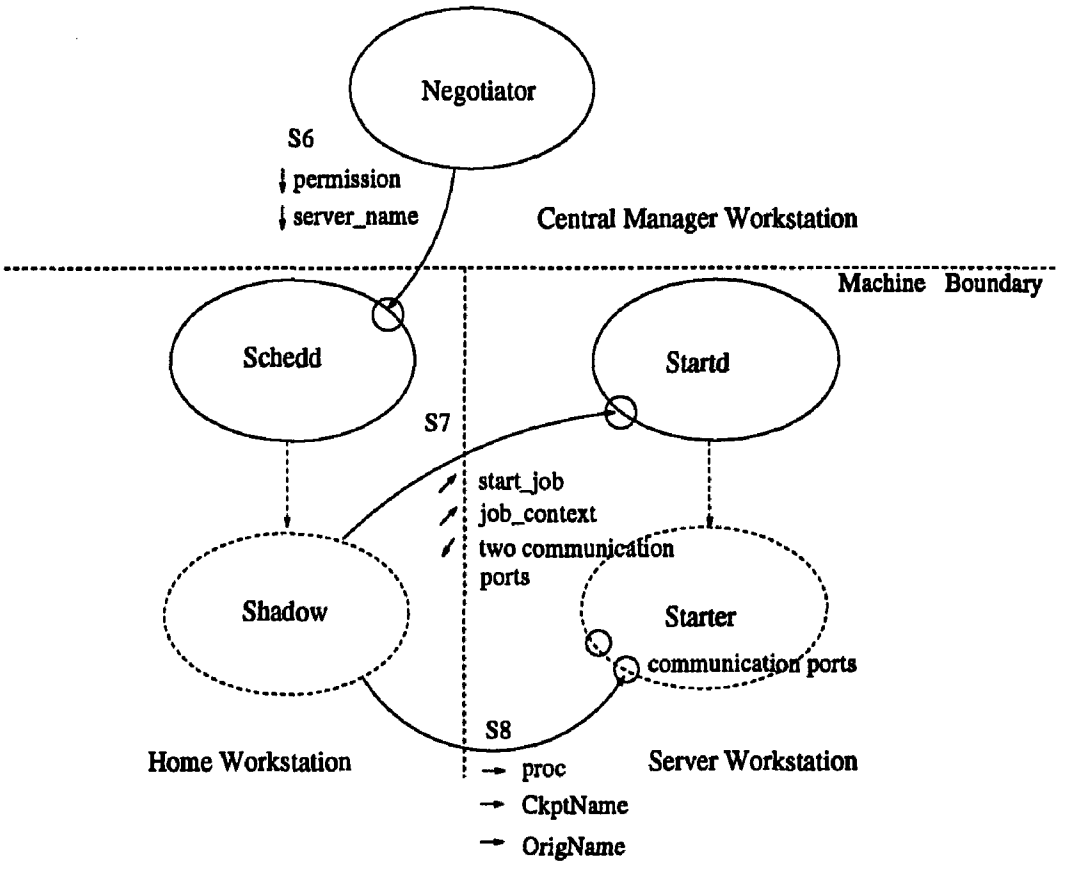
(a) Negotiation process

Figure 8.2: Interactions among Condor daemons.

or locally). Negotiator then contacts each workstation with queued tasks, one at a time, starting with the workstation with the highest priority, and inquires to relocate the task(s) queued on the workstation. If the swap space on the workstation being inquired is enough for Shadow processes[2], the workstation supplies Negotiator with the information on the required OS, architecture, and the task size, of a queued task, with which Negotiator finds a server workstation for the task. A workstation is qualified as a server if (i) both its CPU and keyboards are idle; (ii) it satisfies the task requirement specified; and (iii) no other task is currently running on it. The negotiation process will be repeated for each queued task[3] until either Negotiator finds for all queued tasks their server workstations, or no server can be located (**S4** in Fig. 8.2). At the end of the negotiation process, Negotiator sends back the updated record of machine priorities to Collector (**S5** in Fig. 8.2).

[2]As will be discussed below, each executing task will have associated Shadow processes running on the home workstation.

[3]The tasks in a local queue are also prioritized with respect to the user-specified priority and the order in which they are queued.



(b) Job transfer process

Figure 8.2: (continued) Interactions among Condor daemons.

For each server located, the task transfer process is collaborated on by (a) Negotiator on the central manager workstation, (b) Schedd and Shadow process on the home workstation, and (c) Startd and Starter on the server workstation in the following steps: Negotiator sends a permission message followed by the name of the server workstation to Schedd on the home workstation (**S6** in Fig. 8.2). Schedd on the home workstation then spawns off a Shadow process which connects to Startd on the located server workstation (**S7** in Fig. 8.2) and will henceforth take care of remote system calls[4] from the server workstation.

Startd on the server workstation, upon being notified by Shadow on the home workstation of the task transfer decision, re-evaluates its workload situation and amount of memory space available. If the situation has not changed since the last time Startd

[4]More on remote system calls will be elaborated in Section 8.4.

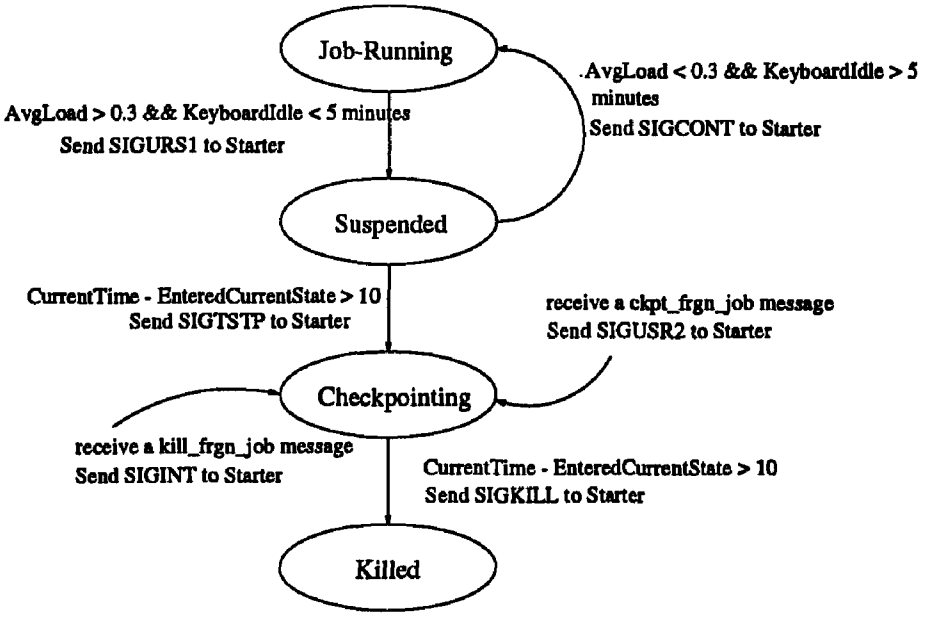


Figure 8.3: **Job_state** transition process.

reported to the central manager, Startd creates two communication ports, and sends the port numbers back to Shadow. Shadow acknowledges the receipt of the port numbers. Startd then spawns off a Starter process (which inherits these communication ports and is responsible for executing the task), and notifies Collector on the central manager of the workload change in the server workstation. Startd henceforth keeps track of **Job_state** of Starter, and signals Starter to suspend, checkpoint, or vacate the executing process whenever necessary to ensure that workstation owners have the workstation resources at their disposal. For example, if during the execution of a task (i.e., **Job_state** is JobRunning), and if either the average workload increases (e.g., **AvgLoad** > 0.3) or the workstation owner returns (e.g., **KeyboardIdle** < 5 minutes), then a SIGUSR1 (suspend) signal is sent to Starter, **Job_state** enters the **Suspend** state, and Starter will temporarily suspend the task. If the task has been suspended for more than a certain period (e.g., 10 minutes), a SIGTSTP (vacate) signal is sent to Starter, Startd enters the **Checkpointing** state, and Starter will abort the task and return the latest checkpoint file to Shadow on the home workstation. Fig. 8.3 gives a complete description of how Startd keeps track of the execution status of Starter and the associated **Job_state** transition process.

The newly–spawned Starter is responsible for (a) getting the executable[5] and other relevant process information from Shadow via either NFS or RPC whichever available and

[5] which is itself a checkpoint file without stack information.

spawning off a child process to execute the task; (b) communicating (via remote system calls) with Shadow on the home workstation for environments/devices–related operations; and (c) suspending, resuming, or checkpointing the executing process upon being requested by Startd (Fig. 8.3). Both Starter and Shadow exit when the task completes/stops execution.

8.3 Incorporation of Distributed LS Policies into Condor

As mentioned in Section 8.1, there are several design drawbacks in Condor:

- the central manager component makes Condor susceptible to a single–workstation crash;
- the information policy periodically invoked introduces a potential bottleneck of network traffic while suffering the effect of using out–of–date state information if the report period is not fine–tuned;
- the location policy is so designed that it is possible for a task arrived at an idle workstation to be transferred to other idle workstations for execution (Section 8.2), since the central manager takes the full responsibility of locating a server workstation.

To remedy the above deficiencies, we eliminate the central manager, and "configure" the functionality of Negotiator and Collector into every participating workstation. Specifically, each participating workstation collects and maintains state information on its own. Moreover, each workstation chooses for every arrived task, if the workstation is not idle, the best server workstation among several candidate workstations, and coordinates with other workstations to reduce the probability of multiple workstations sending their tasks to the same idle workstation with the objective of distributing tasks evenly throughout the system.

8.3.1 LS Policies Used

We now discuss how to incorporate our proposed LS policies into Condor to achieve the above objective:

Transfer Policy: Upon submission/arrival of a task, Schedd on the home workstation determines whether or not the task can be locally executed. That is, the transfer policy is invoked upon arrival of a task, and hence a task transfer, if ever takes place, will occur during an **exec** system call and the new address space will be created on the server workstation. This reduces significantly the process state needed to be transferred. A task is locally executed on the home workstation if **AvgLoad** (the current value of UNIX 1–minute average load)

is less than or equal to 0.3 and the **KeyboardIdle** time (the smallest keyboard idle time observed for all terminals) is greater than 15 minutes, and no other tasks are currently running on the workstation. If the task cannot be locally executed, a transfer decision is made and the location policy is invoked to select a server workstation (if possible) for the task. The workstation rescans its task queue periodically, treats each queued Condor task (i.e., the task which fails to locate a server workstation at the time of arrival) as it were newly–arrived, and repeats the transfer policy.

Information Policy: Region–change broadcasts are used, and message exchanges occur only when the state of a workstation changes significantly (i.e., switches from one state region to another). As discussed in Chapter 3, the communication overheads thus introduced are reduced while the state information kept at each workstation is more likely to be up–to–date. The state defined in our current version is the combination of three quantities: **AvgLoad**, **KeyboardIdle**, and the **State** (NoJob, JobRunning, Suspended, Vacating, or Killed) of the workstation, and the state space is divided into two state regions: runnable and unrunnable. The workstation is said to be in the runnable state region if **AvgLoad** $\leq$ 0.3, **KeyboardIdle** $>$ 15 minutes, and **State** is NoJob. Extension to multiple state regions is conceptually straightforward.

Location Policy: Based on the topological property of the system, each workstation orders all the other workstations into a *preferred list* subject to the properties, **P1** and **P2**, mentioned in Section 3.2. For example, Fig. 8.4 gives the preferred list in a 4–cube system. When a workstation is unable to run a task, it will contact the first "runnable workstation" found in its preferred list, and tries to transfer the task to that workstation. It is important to note that although the preferred list of each workstation is generated *statically*, the actual preference of the workstation in transferring a task may change dynamically with the state of the workstations in its preferred list. That is, if a workstation's most preferred workstation gets unrunnable, this fact will be known to the workstation via a state-region change broadcast and its second preferred workstation will become the most preferred. (It will be changed to the second most preferred whenever the original most preferred becomes runnable, which will be again informed via a state-change broadcast.)

8.3.2 Daemon Configuration

We come up with three daemons, Collector, Schedd, and Startd, which reside constantly on each participating workstation for the decentralized LS mechanism (Fig. 8.5).

Order of preference	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
node 0	1	2	4	8	6	10	12	3	5	9	14	13	11	7	15
node 1	0	3	5	9	7	11	13	2	4	8	15	12	10	6	14
node 2	3	0	6	10	4	8	14	1	7	11	12	15	9	5	13
node 3	2	1	7	11	5	9	15	0	6	10	13	14	8	4	12
node 4	5	6	0	12	2	14	8	7	1	13	10	9	15	3	11
node 5	4	7	1	13	3	15	9	6	0	12	11	8	14	2	10
node 6	7	4	2	14	0	12	10	5	3	15	8	11	13	1	9
node 7	6	5	3	15	1	13	11	4	2	14	9	10	12	0	8
node 8	9	10	12	0	14	2	4	11	13	1	6	5	3	15	7
node 9	8	11	13	1	15	3	5	10	12	0	7	4	2	14	6
node 10	11	8	14	2	12	0	6	9	15	3	4	7	1	13	5
node 11	10	9	15	3	13	1	7	8	14	2	5	6	0	12	4
node 12	13	14	8	4	10	6	0	15	9	5	2	1	7	11	3
node 13	12	15	9	5	11	7	1	14	8	4	3	0	6	10	2
node 14	15	12	10	6	8	4	2	13	11	7	0	3	5	9	1
node 15	14	13	11	7	9	5	3	12	10	6	1	2	4	8	0

Figure 8.4: Preferred list in a 4-cube system.

Similarly as in Condor, two additional processes, Shadow and Starter, run on the home workstation and on the server workstation whenever a task is executed. Note that we carefully configure the transfer, information, and location policies only into Schedd, Startd, and Collector, and retain Shadow and Starter which deal with process transfer, execution, and checkpoint unchanged for the distributed LS mechanism. The functionality of, and the interactions among, daemons are depicted in Fig. 8.6, and are described below.

Collector

Collector is responsible for collecting local workload information, broadcasting a region-change message whenever necessary, updating the workload information of other workstations in its preferred list upon receiving a broadcast message, and responding to Schedd and Startd for information requests.

The local task queue, the average workload (in terms of **AvgLoad**, **KeyboardIdle**, and the **Job_state** of the workstation), and the disk/memory space available are measured upon Collector timeout, or upon receiving a **workload_update** message from the Startd.[6] The parameters measured are then used to evaluate whether or not a workstation is runnable. A workstation is evaluated as runnable (i.e., **Busy** = false) if the function

$$\textbf{START} : (\textbf{AvgLoad} \leq 0.3) \;\&\&\; \textbf{KeyboardIdle} > 15 \text{ minutes}$$

[6]When a task starts or exits/dies, the Startd notifies the Collector to update workload situation.

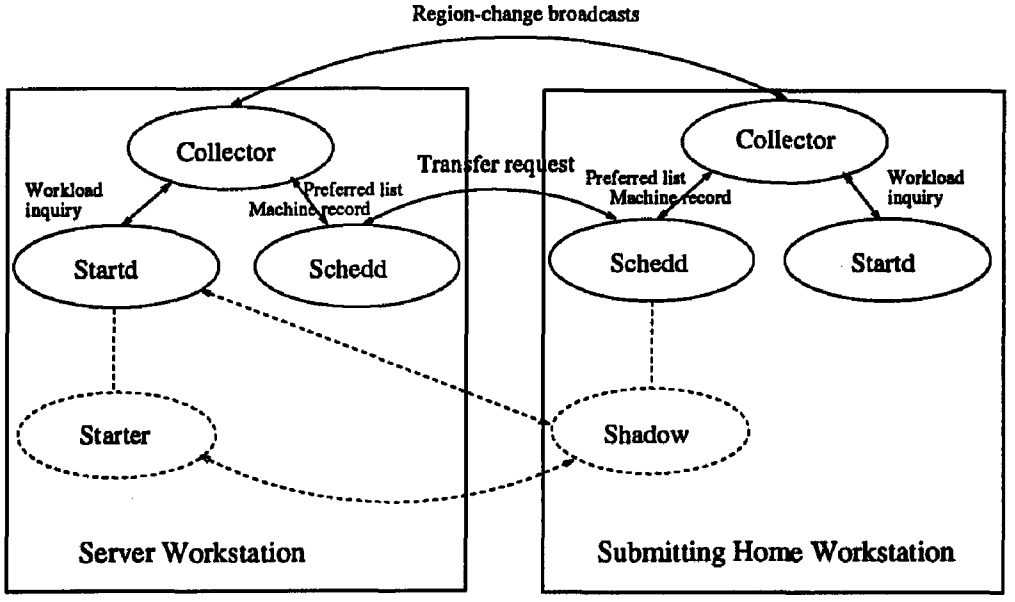


Figure 8.5: Daemons in Modified Condor.

is true and **Job_state** of the workstation is NoJob.

A state-region change message is broadcast to Collectors on other workstations in the preferred list whenever the state switches from from runnable to unrunnable (because of the increase in average workload, return of the workstation owner, or receipt of a task), or vice versa (**S1** in Fig. 8.6). This message is a machine record, the format of which is given in Fig. 8.7. The machine record contains, among other things,

(I1) the hostname, the network address, and the network address type,

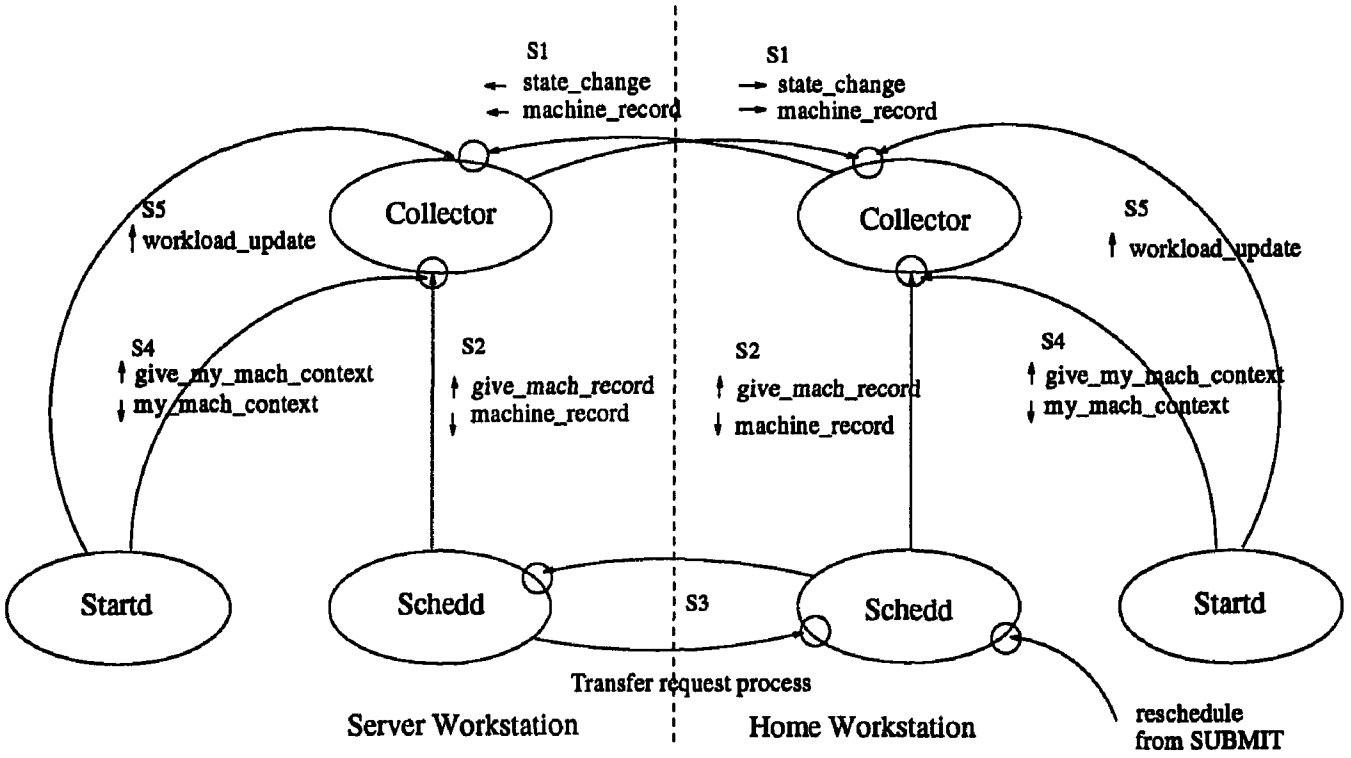
(I2) the indicator variable of whether or not a Condor task is runnable, **Busy**, along with other workload–related parameters, **AvgLoad**, **KeyboardIdle**, **Job_state**,

(I3) the operating system, **OpSys**, and the architecture, **Arch**, of the workstation,

(I4) the swap space, **VirtualMemory**, available in virtual memory, and the disk space, **Disk**, available on the file system where foreign checkpoint files are stored. Note that **VirtualMemory** is only calculated at the time of state–change broadcasts (but not periodically at every timeout), because its calculation is expensive.

(I5) a time–stamp.

As will be clearer later in the discussion of Schedd, **(I3)** is used to verify whether or not a workstation's OS and architecture satisfies the task requirement specified by users; **(I4)** is used to verify whether or not a workstation has enough memory/disk space for

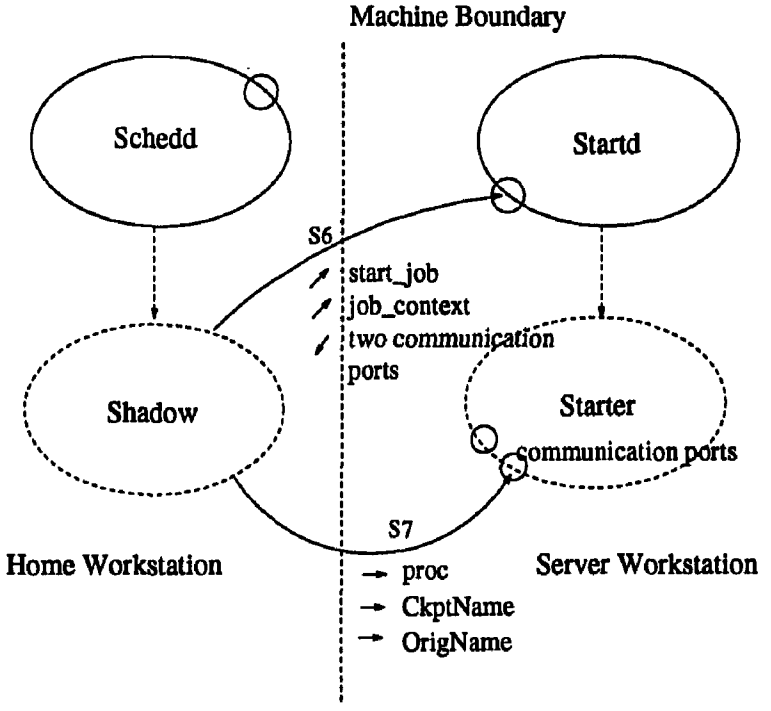


(a) Negotiation process

Figure 8.6: Interactions among daemons in the distributed mechanism.

running foreign tasks; and (**I5**) is used to indicate the degree of a record being obsolete. Upon receiving a state–change broadcast from one of the Collectors on other workstations, the machine record corresponding to the broadcasting workstation in the preferred list is updated.

There are two possible situations Schedd will ask information from Collector (**S2** in Fig. 8.6): (i) when Schedd receives a new task and asks for its own machine record; (ii) when Schedd decides to transfer the task and asks for the machine record of the first runnable workstation available in the preferred list. On the other hand, Startd asks Collector for the machine context which contains workload–related and memory/disk space–related parameters (**S4** in Fig. 8.6), when it wants to check whether a running task should be suspended, checkpointed, resumed, or vacated. (More on this will be discussed in Section 8.3.2.)



(b) Job transfer process

Figure 8.6: (continued) Interactions among daemons in the distributed mechanism.

Schedd

Schedd determines whether or not a (local or remote) task can be executed on the workstation, and, in the case of not executing an arrived task, initiates the location policy to locate a candidate workstation for task transfer. Also, Schedd invokes the location policy periodically for tasks that did not find their servers upon their arrival and are currently queued on the workstation.

There are three major events Schedd handles: the arrival of a task, the receipt of a transfer request, and the periodic timeout:

Upon arrival of a task: upon receiving a reschedule message from the **submit** program, Schedd gets the local machine record from Collector (**S2** in Fig. 8.6), evaluates the parameter **Busy**, and checks whether or not the task requirement is satisfied. The task requirement includes the system configuration (**(I3)** in Section 8.3.2) and the disk/memory space needed for executing the task (**(I4)** in Section 8.3.2).

If the task can be executed locally, a Shadow process is spawned off. Shadow con-

```
typedef struct status_line {
        char    *name;          /* hostname */
        char    *state;         /* state: NoJob, JobRunning, Suspended,
                                   Vacating, Killed */
        float   load_avg;       /* AvgLoad */
        int     kbd_idle;       /* KeyboardIdle */
        char    *arch;          /* Arch */
        char    *op_sys;        /* OpSys */
} STATUS_LINE;

typedef struct mach_rec {
        struct  mach_rec        *next;
        struct  mach_rec        *prev;
        char                    *name;          /* hostname */
        struct  in_addr         net_addr;       /* network address */
        short                   net_addr_type;  /* network address type */
        CONTEXT                 *machine_context;
                                                /* contains VirtualMemory and Disk
                                                   information */
        int                     time_stamp;
        int                     busy;           /* runnable or unrunnable */
        STATUS_LINE             *line;
} MACH_REC;
```

Figure 8.7: Data structure for machine record.

tacts the local Startd which then creates two communication ports, sends the port numbers back to Shadow, and spawns off a Starter. Thus, Starter inherits the two communication ports and shall actually execute the task. Shadow and Starter then communicate through the communication ports, and the task execution/checkpoint process proceeds as in Condor. Note that by carefully "reconfiguring" the daemons, we leave the "low–level" implementation mechanism for task execution and checkpoint unchanged in our distributed LS mechanism.

If the task cannot be executed locally (either **Busy** is true, or the task requirement is not satisfied), then Schedd checks if there is enough swap space for a new Shadow process. If the swap space is not enough, the task is queued and will be attempted for execution/transfer upon next scheduled timeout. If the swap space is sufficient, Schedd gets from Collector the machine record of the first runnable workstation in the preferred list, and checks whether or not the task requirement can be satisfied on that workstation. If not, the machine record of the next runnable workstation available in the preferred list is fetched from Collector and checked against the task requirement. The process repeats itself until either a target server workstation is found or the preferred list is exhausted. In the latter case, the task is queued for later execution/transfer attempts.

If a target server workstation is located, Schedd sends a transfer request to Schedd

on the target server workstation (**S3** in Fig. 8.6). Either a **transfer_ok** or a **transfer_not_ok** message will be received from the target server workstation, depending on whether or not the target workstation is truly runnable: if a **transfer_ok** message is received, a Shadow process is spawned off on the home workstation which notifies the Startd on the target server workstation of its responsibility to execute the task. If the workload situation has not changed on the target server workstation since its last region–change broadcast, a **startd_ok** message, along with two communication ports, is received. The communication and task transfer/execution operations between Shadow and Starter then proceed as in Condor. If the workload situation has changed and is not runnable anymore, a **startd_not_ok** message is received, under which case Schedd gets from Collector the machine record of the next runnable workstation available in its preferred list, and repeats the transfer–request process until either a target server workstation is found or the preferred list is exhausted. On the other hand, if a **transfer_not_ok** message is received, Schedd gets from Collector the machine record of the next runnable workstation, and repeats the transfer–request process as described above.

To deal with a possible machine failure, the ioctl system call is used to designate the sockets as non–blocking: an I/O request that cannot be completed is not performed, and return is made immediately. Moreover, a timer is set for each connection: if no response has ever come back until the timer expires, return is also immediately made. In either case, Schedd repeats the transfer–request process for the next runnable workstation available in the preferred list.

Upon receipt of a transfer request: upon receiving a transfer request, Schedd gets from Collector the local machine record and evaluates the function **Busy**. In terms of the four–component task requirements, Schedd needs only to check **VirtualMemory**, because

- **OpSys** and **Arch** are already checked by the home workstation who initiates the transfer request;
- The **Disk** space available under the directory where checkpoint files are saved will not change if no task is executing on the workstation. So, it suffices to assure the **Disk** space has not changed by checking if the workstation is non–**Busy**;
- Since **VirtualMemory** is calculated at the time of state-region change broadcast, the **VirtualMemory** information collected (via state–change broadcasts) by the requesting workstation may differ from the actual **VirtualMemory** information currently kept if either a broadcast message is lost or not yet received by the requesting work-

station before the transfer request was made. Hence, **VirtualMemory** needs to be re-checked.

If **Busy** is false and **VirtualMemory** is enough, Schedd responds with a **transfer_ok** message. The Shadow process on the requesting workstation will then contact Startd on the server workstation (which honors the transfer request) to handle the "low-level" mechanism of task execution/transfer and checkpoint process. Otherwise, the Schedd replies a **transfer_not_ok** message.

Upon scheduled timeout: upon scheduled timeout, Schedd first prioritizes the tasks currently queued on the local workstation based on their user-specified priority, queueing time, and whether or not a task was ever executed. Higher priority is given to tasks with higher user-specified priority, longer queueing time and/or tasks which were vacated from server workstations because of the return of the server workstation owner or some abnormal situation on the server workstation. Schedd then initiates the location process for each queued task, starting from the task with the highest priority.

Startd

Upon being notified by a Shadow process of the responsibility to execute a task, Startd generates two communication ports, spawns off a Starter to execute the task, keeps track of the execution status of the task, and signals the Starter, whenever necessary, to suspend, resume, checkpoint, or vacate the executing task. There are five events Startd will handle: the receipt of a **start_task** message from the Shadow on a requesting workstation, the receipt of a SIGCHLD signal (at the exit of Starter), the periodic starter timeout, the receipt of a **checkpoint_task** message from Shadow on the home workstation, and the receipt of a **kill_task** message from Schedd on the home workstation.

Upon receipt of a start_task message: upon receiving a **start_task** message from a requesting Shadow, Startd gets from Collector its machine context (**S4** in Fig. 8.6), and re-evaluates the **Busy** function. If the **Busy** function is false, two communicating ports are created and returned (along with a **startd_ok** message) to the Shadow on the requesting home workstation. Startd then waits for connection from Shadow to these two ports. When this connection is made, Startd spawns off a Starter, closes the two communication ports, changes the **Job_state** of the workstation to JobRunning, and notifies Collector of its state_change (**S5** in Fig. 8.6; in which case Collector updates workload). If the **Busy** is true, a **startd_not_ok** message is returned.

Upon receipt of a SIGCHLD signal: upon receiving a SIGCHLD signal, Startd clears up the checkpoint files in the directory where the checkpoint files are stored, changes the **Job_state** of the workstation to NoJob, and notifies Collector of its state_change (**S5** in Fig. 8.6).

Upon periodic startd timeout: upon periodic Startd timeout, Startd gets from Collector the parameters **AvgLoad** and **KeyboardIdle** (specified in the machine_context, **S4** in Fig. 8.6), and properly signals Starter based on these workload–related parameters to assure that workstation owners have the workstation resources at their disposal (Fig. 8.3).

Upon receipt of a checkpoint_Job or a kill_task message: Upon receiving a checkpoint_task (kill_task) message from Shadow (Schedd), Startd sends a SIGUSR2 (SIGINT) signal to Starter, and enters the **Checkpointing** state (Fig. 8.3).

8.4 Implementation Issues

In this section, we discuss how we handle some of the implementation issues, such as where to place the LS mechanism (inside or outside the OS kernel), how to transfer process state (virtual memory, open files, and process control blocks) during task transfer/migration, and how to support location transparency and reduce the effects of residual dependency.

Where the LS mechanism is located: We follow Condor's principles, and implement the LS mechanism outside the OS kernel in trusted daemon processes. Placing the mechanism outside the kernel incurs execution overhead and latency (e.g., in the form of kernel calls) in passing statistics (from kernel to daemon processes) and LS decisions (in the other direction). However, as discussed in [AF89], the dominating factor in assessing LS performance lies more in the global communication overhead and aggregate resource management than in (small) delays incurred by kernel calls. Moreover, placing the mechanism outside the kernel facilitates later expansion or generalization of our other LS strategies to deal with large communication latency [SH91], excessive task transfer [HS91], and node/link failure [HS93b, SH93, CS91]. One inherent limitation resulted from placing the LS mechanism outside the OS kernel is that inter–process communication and signal facilities cannot be easily implemented, and are not supported in the current implementation.

Approach to transferring process state: Process state typically includes the virtual memory, the open files, message channels, and other kernel states contained in the process

control block. In Condor, the state of a process is transferred in the form of checkpoint files. Before a process is executed for the first time, its executable file is augmented to a checkpoint file with no stack area, so that every checkpoint file is henceforth handled in the same way. Moreover, every process is periodically checkpointed, and a new checkpoint file is created from pieces of the previous checkpoint (which contains the text segment) and a core image (which contains the data and stack segments) as follows: the LS mechanism causes a running task to checkpoint by sending it the signal SIGTSTP. When a task is linked, a special version of "crt()" is included which sets up CKPT() as the SIGTSTP signal handler. Information about all open files which the process currently has is kept in a table by the modified version of the **open** system call routine. When CKPT() is called, it updates the table of open files by seeking each one to the current location and recording the file position. Next a **setjmp** is executed to save key register contents (e.g., stack pointer and program counter) in a global data area, then the process sends itself a SIGQUIT signal which results in a core dump. Starter then combines the original executable file, and the core file to produce a checkpoint file.

When the checkpoint file is restarted, it starts from the special "crt()" code, and it will set up the restart() routine as a SIGUSR2 signal handler with a special signal stack (in the data segment), then send itself the SIGUSR2 signal. When restart() is called, it will operate in the temporary stack area and read the saved stack in from the checkpoint file, reopen and reposition all files, and execute a **longjmp** back to CKPT(). When the restart routine returns, all the stacks have been restored, and CKPT() returns to the routine which was active at the time of the checkpoint signal, not "crt()".

Location transparency and residual dependency: *Location transparency* is one of the most important goals in implementing load sharing. By transparency, we mean a process's behavior should not be affected by its transfer. Its execution environment should appear the same, it should have the same access to system resources such as files, and it should produce exactly the same results as if it had not been transferred [AF89, DO91]. To maintain location transparency, sometimes the home workstation has to provide data structure or functionality for a process after the process is transferred from the workstation [DO91]. This need for a home workstation to continue to provide some services for a process remotely-executed is termed as *residual dependency*. In our implementation, location transparency is achieved at the expense of residual dependency in the following manner: the LS mechanism preserves the home workstation's execution environment for the remote process by using "remote system calls" in which requests for file/device access are trapped

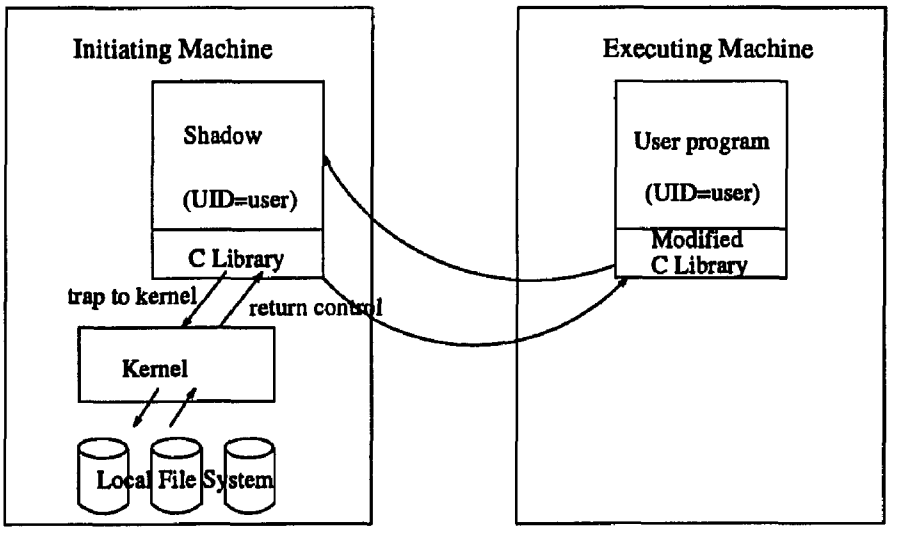


Figure 8.8: Remote system calls.

and forwarded to the Shadow process on the home workstation. As was discussed in Section 8.3, whenever a workstation is executing a task remotely, it also runs a Shadow process on the home workstation. The Shadow acts as an agent for the remotely executing task in doing system calls. Specifically, each task submitted to the LS mechanism is linked with a special version of the C library. The special version contains all of the functions provided by the normal C library, but the system call stubs have also been changed to accomplish remote system calls. The remote system call stubs package up the system call number and arguments and send them to the Shadow via the network. The Shadow, which is linked with the normal C library, then executes the system call on behalf of the remotely running task in the normal way. The Shadow then packages up the results of the system call and sends them back to the system call stub (in the special C library on the submitting machine) which then returns its result to the calling procedure (Fig. 8.8).

8.5 Related Work

In this section, we review several existing load sharing mechanisms, i.e., the V-system [TLC85, Stu88], the Sprite OS [DO91], the Charlotte OS [AF89], and the Condor software package [LLM88, LL90], with respect to their design policies and implementation features.

8.5.1 Design Policies

As discussed in Chapter 1, the design of LS mechanisms is characterized by the transfer, location, and information policies. We now discuss how the existing systems tailor their LS policies.

V–System: The V–system [TLC85, Stu88] uses a state-change driven information policy. Each node broadcasts its state whenever its state changes significantly. State information consists of expected CPU and memory utilization and specification about the machine itself, such as its processor type. The broadcast state information is cached by all the nodes. If the distributed system is large, each machine can cache information about only the best N nodes, e.g., those nodes having idle or underloaded CPU and memory.

The V–system's transfer policy selects only newly–arrived tasks for transfer. Its transfer policy defines a node as a receiver if it is one of the M most lightly–loaded nodes in the system, and as a sender if it is not. Its (decentralized) location policy locates receivers as follows: when a task arrives at a machine, it consults the local cache and constructs the set containing the M most lightly–loaded machines that can satisfy the task's requirements.[7] If the local machine is one of the M machines, then the task is scheduled locally. Otherwise, a machine is chosen *randomly* from the set and is *polled* to verify the correctness of the cached data. Note that the random selection cannot totally avoid the possibility of multiple machines selecting the same remote machine for task transfer. If the cached data matches the machine's state, the polled machine is selected for executing the task. Otherwise, the entry for the polled machine is updated and the polling procedure is repeated.

Sprite: The Sprite OS [DO91] uses a centralized state–change driven information policy. Each workstation runs a background process called the *load–average* daemon, which monitors the usage of that machine. When the workstation appears to be idle, the load–average daemon notifies a *central migration server* that the machine is ready to accept migrated processes. A threshold–based rule is used to decide whether or not a workstation is idle: when the workstation has had no keyboard or mouse input for at least 30 seconds and the number of active tasks is less than the number of processors at the workstation, it is considered idle.

The transfer policy used in Sprite takes place in three cases. In the first case, tasks may be chosen *manually* by users for remote execution. The transfer policy is thus

[7]The V–system's load index is the CPU utilization at a node. To measure CPU utilization, a background process that periodically increments a counter is run at the lowest priority.

not completely automated, and tasks are submitted for process migration through two application programs, pmake and mig. Another case arises if foreign tasks executing at a workstation must be evicted (and migrated back to their home workstations) to ensure the availability of the workstation's resources to the workstation owner. The third case is invoked to ensure fair allocation of computing resources. If the centralized server cannot find an idle workstation for a remote execution request and it finds that a process has been allocated more than its fair share of workstations, then the server reclaims one of the workstations being used by that process and evicts that process. The freed workstation is then granted to the remote execution request.

The location policy is centralized: to locate a receiver, a workstation contacts the central migration server (through a standard library procedure) which then selects an idle workstation at random.

Charlotte: In Charlotte [AF89], transfer and location policies are dictated by Starter utility processes which base their decisions on statistical information provided by the kernels they control and on state information they exchange among themselves. One can introduce various transfer/location policies into the Starter utility processes by customizing and invoking the policy procedure. The policy procedure can choose to request some source kernel to undertake process migration. Starter-to-Starter negotiation begins when a process migration operation is invoked and may result in either a decision to migrate a process or a rejection. In addition, Starters accept advice from privileged utilities (to allow manually-directed transfers and to enable/disable automatic control).

The information policy is realized by a statistician thread which awakens periodically to sample, average, and report statistics to the Starter. A summary of state information is then periodically exchanged among the Starter utility processes.

Condor: Condor is centralized with the central manager acting as a task dispatcher. To submit a task, a user links it with a special system–call library and places it in a local queue of background tasks. The transfer policy is invoked either upon submission of a task or upon a scheduled timeout. In both cases, the central manager finds idle workstations not only for the submitted task but also for tasks queued on all participating workstations. To accomplish this, Condor uses a *periodic* information policy. Each workstations reports to the central manager periodically the state information on the average workload index, the keyboard/mouse idle time and the task queue situation. A workstation is considered idle if the owner has not been active for at least 12.5 minutes. For each waiting task, the central

manager negotiates with workstations reported to be idle using the location policy described in Section 8.2, and if the negotiation succeeds, it transfers the task to that workstation.

To ensure that workstation resources are available to the owner of the workstation, the Startd at each workstation checks for local activity from the owner every 30 seconds if a foreign background task is being served at that workstation. If the owner has been active since the previous check, the Startd preempts the foreign task and saves its states. If the workstation owner remains active for 5 minutes or more, the foreign task is preemptively transferred back to the home workstation and returned to the waiting queue (Fig. 8.3). The task may be transferred later, upon a scheduled timeout, to an idle workstation if one such workstation is located by the central manager.

8.5.2 Implementation Issues

As discussed in Sections 8.1 and 8.4, two issues relevant to LS implementation are where to locate the LS mechanism and how to transfer process state. We now discuss how the existing systems deal with these issues.

Where to place the LS mechanism: The principal reason to place the LS mechanism in the kernel is to reduce execution overhead and latency. However, integrating the LS mechanism in the kernel might make the kernel less modularized and make later modification or generalization difficult. The LS mechanism in the V-system and Sprite OS resides in the kernel, and is integrated (and closely interplays) with process scheduling, memory management, and interprocess communication. Condor, on the other hand, is implemented completely outside the kernel. Charlotte, being somewhere in between, places the LS policies in *utility* processes outside the kernel and the low-level process migration facilities inside the kernel.

Strategies for transferring process state: Process state typically includes virtual memory, open files/message channels, and kernel states (maintained in process control blocks). For virtual memory transfer, Charlotte sends the entire memory image of the process to the destination workstation at the transfer/migration time. The transfer may take seconds, during which the process is frozen for execution on either the home or the destination workstation. To reduce the long freeze time, the V-system uses the *pre-copying* scheme where the process is allowed to continue executing while the virtual memory is transferred. The process is only frozen when some pages are modified on the home workstation after they have been copied to the destination workstation, in which case the modified pages

are then re–copied. In Sprite, the home workstation freezes the process, flushes its dirty pages to backing files (which is backing storage for virtual memory and is accessible through the network file), and discards its address space. On the destination workstation, the process starts executing with no resident pages and uses the standard paging mechanisms to load pages from backing files as they are needed.

In message–based systems such as Charlotte and V, all interactions of a process with the rest of the system occur through message channels. Once the basic execution state of a process has been migrated, all the remaining issues can be solved simply by forwarding message on the process's message channels. The operating system can use the *arranging for forwarding* approach [DO91], and need only to change the sender and receiver addresses so that message to and from the channel can find their way from and to the process. In a system such as Sprite that is based on kernel calls, an open file is attributed by three components of the process state: a file reference, caching information, and an access position, all of which must be properly transferred.

The other state consists almost entirely of fields from the process control block (which is not bulky and does not involve distributed/shared states), and all three systems transfer these fields to the destination workstation and reinstate them in the process control block on the destination.

8.6 Conclusion and Current Status

We discussed a preliminary implementation of our decentralized LS mechanism proposed in Chapter 3 based on the Condor software package. We removed the central manager in Condor, and incorporated the functionality of the central manager into every participating workstation. Each participating workstation collects state information on its own via region–change broadcasts, and makes LS decisions based on the state information collected. The probability of multiple machines sending their tasks to the same idle machine is minimized by using the concept of preferred list in the location policy.

Special care has been taken to fuse our decentralized LS policies into the existing Condor software so as to require as little modification as possible is needed. The remote system call and process checkpoint facilities in Condor are adopted to provide location transparency, to preserve the home workstation's execution environment, and to transfer the state of a process.

At the writing of the dissertation, the initial version of the decentralized LS mechanism is being implemented/tested, and initial performance measurements are expected to

be done shortly. We also plan to incorporate other features proposed in Chapters 3–6 into the LS mechanism, and equip the LS mechanism with the abilities to deal with large communication latencies, excessive task transfers and task collisions, and component failures.

CHAPTER 9

DISCUSSION AND FUTURE WORK

In this chapter we recapitulate the contributions of the dissertation, and explore possible extensions and future directions for the work presented here.

9.1 Research Contributions

We recognize that real-time task management is an important problem in distributed real-time systems. The general methodology for managing a real-time task system in a distributed environment consists of four phases, and we have focused on the allocation of periodic task modules and the redistribution of non-periodic tasks.

In particular, we have proposed a module allocation algorithm (**MAA**), based on the branch and bound method, to find a module allocation that maximizes the probability of completing all periodic tasks with both logical and timing correctness. The **MAA** not only assigns task modules to PNs, but also uses a module scheduling algorithm (**MSA**) of polynomial-time complexity to schedule all modules assigned to each PN so that all periodic tasks are ensured to be completed by their deadlines [HS92]. Moreover, in order to speed up the branch-and-bound process, a dominance relation is derived from the requirement of timely completion of tasks and used in the branching process, and an upper bound for the objective function is derived for every partial allocation and used to prune intermediate vertices in the bounding process. As our extensive simulation study demonstrated, the **MAA** always finds the best allocation at tractable computational costs for task systems with less than 50 modules and/or distributed systems with less than 40 PNs.

We have designed an effective LS mechanism to enable underloaded/capable nodes in a distributed real-time system to share the loads of overloaded/incapable ones so that

(1) the probability of dynamic failure, P_{dyn}, is minimized. We used the transfer policy of *dynamic* threshold type to determine a node's capability, *region-change broadcasts* to reduce the communication overhead while keeping the state information as updated

as possible, and the *preferred list* structure to minimize the probability of multiple nodes sending their overflow tasks to the same underloaded node [SH91].

(2) the undesirable effect of communication delays on the consistency of state information is alleviated [SH91]. An important feature of this design is the use of *time-stamped region-change broadcasts* and *prior/posterior distribution* of (load) state to characterize the inconsistency between the state a node observes and the true systemwide state, and Bayesian decision theory to estimate the true states of other nodes based on out-of-date state information. We have shown the performance of the LS mechanism with Bayesian decision to be less susceptible to communication delays in state collection and task transfers.

(3) the probability that a remote node fails to complete a transferred task because of future arrivals of tighter-laxity tasks is reduced [HS91]. The probability that a remote node fails to complete a transferred task in time because of future tighter-laxity tasks that arrived after the arrival of the transferred task but prior to the execution of the transferred task is approximately derived using queueing analysis, and is figured into the LS decision. All parameters needed for calculating the probability distribution of interest are on-line collected/estimated with Bayesian estimation. Our simulation results have shown that with consideration of future task arrivals, the occurrence of task collisions and excessive task transfers is significantly reduced. We have also shown that the performance degradation of the LS mechanism (with the use of Bayesian estimation) due to statistical fluctuations in task arrivals is tolerable within a wide range of bursty task arrival patterns.

(4) node failures are detected in a timely manner by a simple timeout mechanism with on-line adjustable timeout periods [HS93b]. We formulate the problem of determining the 'best' timeout period as a hypothesis testing problem with the objective of maximizing the probability of detecting node failures subject to a pre-specified probability of false alarm. Our simulation results show that the LS mechanism which combines the timeout mechanism, and a few extra, timely broadcasts can significantly reduce the probability of missing task deadlines, as compared to other schemes either without any timeout mechanism or with a fixed timeout period.

We have developed semi-Markov process models to assess analytically the performance of our LS mechanism as well as other existing LS schemes [SH92, SH93]. The analysis methodology presented in this dissertation is quite general in the sense that it can be extended to other LS mechanisms and other underlying communication subsystems. For

example, extending the methodology to other LS mechanisms, one needs only to properly derive the parameters that characterize the transfer policy and location policy, respectively. Once they are determined, the derivation of other metrics follows the same approach.

We have also discussed how to implement, based on Condor, the LS mechanism that incorporates the concept of buddy set, preferred list, and region–change broadcasts as a prototype software layer that sits on top of the OS (e.g., BSD 4.3 UNIX in our current implementation) and provides the transparent LS service to application programs [HS93c]. Both the analytic and simulation results and the experimental measurements have indicated that by using judicious exchange/use of state information and Bayesian decision mechanisms, the LS mechanism makes a significant improvement in minimizing P_{dyn} over those simple LS schemes. This is in sharp contrast to the common notion that simple LS schemes yield performance close to that of complex ones for general–purpose systems where minimizing the mean response time is the main concern. Since missing a task deadline can cause a disastrous accident in a real–time environment, a more complex, but intelligent, LS mechanism should be employed to minimize P_{dyn}.

9.2 Future Directions

This work has revealed several promising research issues that are worth further investigation:

Experimentation with, and Enhancement of, Implemented LS Mechanisms: Expanding the initial version of the implemented prototype LS mechanism will allow us to conduct several interesting experiments. For example, we will be able to measure the performance improvement with the consideration of future tighter–laxity task arrivals and with the incorporation of a timeout mechanism in case of component failure. We will also be able to measure the overheads of different components in our LS mechanisms, and experimentally analyze the tradeoffs between the associated complexity and the resulting benefit. Besides, we can also experiment on different local scheduling disciplines to assess their effects on LS performance.

Incorporation of Fault Tolerance into LS: We have addressed the issue of detecting node failures in this dissertation: we incorporated a timeout mechanism with an on–line adjustable timeout period into LS. The LS mechanism can be enhanced further to handle node failures. For example, in Chapter 3 we incorporated the concept of preferred list [SC89a] — ordering fault–free nodes as preferred receivers of overflow tasks — into our

LS mechanism. In Chapter 4, we proved that if the preferred list is structured so that the features of uniqueness and symmetry are retained then the probability of more than one nodes simultaneously sending their tasks to the same node is minimized. Moreover, overflow tasks are evenly distributed among capable nodes. One potential problem is that the occurrence/detection of node failures will destroy the original structure of a preferred list if the node diagnosed as failed are simply dropped from the list. Moreover, if a node fails before all the tasks in its queue are completed, those tasks will be lost unless some fault-tolerant mechanism is provided. Thus, it is desirable to (1) develop an algorithm to modify the preferred lists in case of node failures in order to retain the desirable features of the preferred list; and (2) devise an approach to coordinate nodes to keep backup copies of tasks arrived in the systems in an effective manner.

Integration of Real–Time Communications Subsystem into LS: As addressed in Chapter 7, no matter which information or location policy is used, we must consider the underlying real–time communication subsystem that supports all LS–related communications activities. That is, a communication subsystem should be designed to support time–constrained communication and incorporated into the proposed real–time LS mechanism for task transfers and state–change broadcasts. Qualitatively, the communication subsystem should be able to

(1) deliver messages/tasks within certain deadline constraints;

(2) support broadcast facilities in the physical or data link layer;

(3) offer services such as maintenance of a global time–base (used for time–stamped region–change broadcasts), and support for group communications.

Our colleagues at Michigan have laid out a framework to handle time–constrained messages based on a communication abstraction called a *real–time channel* [KS91a, ZS92]. They have also developed a broadcast algorithm [KS91b] and a clock synchronization scheme [RKS90]. Our overall goal is to implement these schemes, along with some of the functionalities of the proposed LS mechanism, in a prototype software that shall be run on the communication subsystem.

Investigation of Underlying Services Provided to LS: In conjunction with the design/implementation of an experimental LS mechanism, one can (i) investigate basic research issues related to the services which the underlying communication subsystem and OS should provide to the LS mechanism, e.g., synchronizing network time, reliable broadcasts,

remote procedure calls, directory service, and security, and propose design solutions; (ii) evaluate analytically the resulting performance of the proposed solutions; and (iii) proceed to build up a layer of service routines that sits on top of the operating system and networking transport services, and offers its services in the form of library procedure calls to applications above (e.g., the LS mechanism), based on the DCE (distributed computing environment) concepts proposed by the OSF.

APPENDIX A

VERIFICATION OF FLOW CONSERVATION

To verify the correctness of Eq. (4.15), one has to show that flow conservation holds for the system with the random LS scheme.

Corollary 1 *For the random LS scheme,* $\sum_{k=0}^{\infty} \alpha(k) p_T(k) = \beta(k)$.

Proof:

$$\begin{aligned}
\sum_{k=0}^{\infty} \alpha(k) p_T(k) &= \sum_{k=0}^{\infty} \Big(\sum_{i=1}^{E_{max}} \sum_{j=0}^{k-1} \lambda q_{ij} \Big) p_T(k) \\
&= \sum_{i=1}^{E_{max}} \sum_{j=1}^{\infty} \sum_{k=j+1}^{\infty} \lambda q_{ij} p_T(k) \\
&= \sum_{j=1}^{L_{max}} \lambda \sum_{i=1}^{E_{max}} q_{ij} \sum_{k=j+1}^{\infty} p_T(k) \\
&= \sum_{j=1}^{L_{max}} \lambda \hat{q}_j \gamma_{j+1} \\
&= \beta(k),
\end{aligned}$$

where the second equality follows from interchanging the summation indices while preserving the range of summation appropriately, and the third equality follows from $q_{ij} = 0$ for $j \geq L_{max} + 1$. □

To verify the correctness of Eq. (4.16), one has to show that flow conservation holds for the system with the proposed LS scheme, i.e.,

Corollary 2 *When those tasks being rejected are not considered,*

$$\sum_{k=0}^{\infty} \beta(k) p_T(k) = \sum_{k=0}^{\infty} \alpha(k) p_T(k),$$

for the proposed scheme.

Proof:

$$\begin{aligned}\sum_{k=0}^{\infty}\beta(k)p_T(k) &= \sum_{k=0}^{\infty}\sum_{j=k}^{L_{max}}\lambda\hat{q}_j\gamma_{j+1}\frac{1-\gamma_{j+1}^{N_B}}{1-\gamma_{j+1}}p_T(k)\\ &= \sum_{j=0}^{L_{max}}\lambda\hat{q}_j\gamma_{j+1}\frac{1-\gamma_{j+1}^{N_B}}{1-\gamma_{j+1}}\sum_{k=0}^{j}p_T(k)\\ &= \sum_{j=1}^{L_{max}}\lambda\hat{q}_j\gamma_{j+1}(1-\gamma_{j+1}^{N_B}).\end{aligned}$$

However, from Corollary 1,

$$\sum_{k=0}^{\infty}\alpha(k)p_T(k) = \sum_{j=1}^{L_{max}}\lambda\hat{q}_j\gamma_{j+1}.$$

Inconsistency results from the nonzero probability of dynamic failure, since the difference, $\gamma_{j+1}^{N_B}$, is the probability that a task with laxity j will fail to complete in time. If these failed tasks are not rejected or continuously transferred from node to node, thus conserving task flow, and/or the probability of dynamic failure is negligibly small, the summation in Eq. (4.16), $1+\gamma_{j+1}+\gamma_{j+1}^2+\ldots+\gamma_{j+1}^{N_B-1}$ can be replaced by $\sum_{k=0}^{\infty}\gamma_{j+1}^k = 1/(1-\gamma_j)$, thus $\beta(T) = \sum_{j=T}^{L_{max}}\lambda\hat{q}_j\gamma_{j+1}/(1-\gamma_{j+1})$, and $\sum_{k=0}^{\infty}\beta(k)p_T(k) = \sum_{k=0}^{\infty}\alpha(k)p_T(k)$. □

To verify the correctness of Eq. (4.18), one has to show that flow conservation holds for the system with the quasi-perfect LS LS scheme.

Corollary 3 *Without considering those tasks being rejected and declared not to meet their deadlines, we have*

$$\sum_{k=0}^{\infty}\lambda(k)p_T(k) = \lambda$$

for the quasi-perfect LS scheme.

Proof:

$$\begin{aligned}\sum_{k=0}^{\infty}\lambda(k)p_T(k) &= \sum_{k=0}^{\infty}\sum_{j=k}^{L_{max}}\lambda\hat{q}_j\frac{1-\gamma_{j+1}^{K_{pn}}}{1-\gamma_{j+1}}p_T(k)\\ &= \sum_{j=0}^{L_{max}}\lambda\hat{q}_j\frac{1-\gamma_{j+1}^{K_{pn}}}{1-\gamma_{j+1}}\sum_{k=0}^{j}p_T(k)\\ &= \sum_{j=0}^{L_{max}}\lambda\hat{q}_j(1-\gamma_{j+1}^{K_{pn}})\\ &\simeq \sum_{j=0}^{L_{max}}\lambda\hat{q}_j \text{ if } \gamma_{j+1}^{K_{pn}}\simeq 0\\ &= \lambda.\end{aligned}$$

Inconsistency again results from the nonzero probability of dynamic failure. If this probability (or $\gamma_j^{K_{pn}}$) is negligibly small and/or the failed tasks are continuously transferred among nodes, task flow will be conserved. □

APPENDIX B

SUMMARY OF RANDOMIZATION TECHNIQUE

We summarize some important results of the randomization technique.

Consider a continuous–time Markov chain $\{X(t), t \geq 0\}$ with generator matrix Q on a finite state space S of size $N+1$. For notational convenience, we enumerate the elements of S as $0, 1, \ldots, N$. Let $\Lambda \triangleq \max_{0 \leq i \leq N} q_i$, then there exists a discrete–time Markov chain $\{Y_n, n = 0, 1, \ldots\}$ and a Poisson process $\{N(t), t \geq 0\}$ with rate Λ, which are independent of each other, such that the process $\{Y_{N(t)}, t \geq 0\}$ has the same finite dimensional distributions as, and is thus probabilistically identical to, $\{X(t), t \geq 0\}$. In the equivalent process, the transition rate from state i is Λ, but only the fraction q_i/Λ of transitions are real and the remaining fraction $1 - \frac{q_i}{\Lambda}$ are fictitious transitions. In other words, $\{X(t), t \geq 0\}$ can be considered as a process which spends a time with an exponential rate Λ in state i and then makes a transition to state j with probability $\mathcal{P}_{ij}$, where

$$\mathcal{P}_{ij} = \begin{cases} 1 - \frac{q_i}{\Lambda} & \text{if } j = i, \\ \frac{q_{ij}}{\Lambda} & \text{if } j \neq i. \end{cases} \tag{B.1}$$

The transient probabilities, $P(X(t) = i)$, $0 \leq i \leq N$, of the continuous–time Markov chain $\{X(t), t \geq 0\}$ can now be easily obtained by conditioning on $N(t)$, the number of transitions in $(0, t]$, i.e.,

$$\begin{aligned} P(X(t) = i) &= P(Y_{N(t)} = i) = \sum_{n=0}^{\infty} P(Y_{N(t)} = i \mid N(t) = n) \cdot P(N(t) = n) \\ &= \sum_{n=0}^{\infty} P(Y_n = i) e^{-\Lambda t} (\Lambda t)^n / n!. \end{aligned} \tag{B.2}$$

In other words, a continuous–time Markov chain $\{X(t), t \geq 0\}$ on a finite state space S, after its randomization, can be viewed as a discrete–time Markov chain, $\{Y_n, n = 0, 1, \ldots\}$, subordinated to a Poisson process $\{N(t), t \geq 0\}$, and thus the transient probabilities can be easily computed using the discrete–time Markov chain.

APPENDIX C

LIST OF SYMBOLS

Notation Used in Chapter 2

P_{ND}: the probability of no dynamic failure, i.e., the probability of all tasks making their deadlines.

P_{ND1}: the probability that all tasks within a planning cycle are completed before their deadlines.

P_{ND2}: the probability that all PNs are operational during the execution of modules assigned to them, and the communication links between communicating PNs are operational for all the intermodule communications that use these links.

N_T: the total number of periodic tasks in the system.

t_i: the invocation time of a periodic task T_i.

p_i: the period of a periodic task T_i.

d_i: the deadline of a periodic task T_i.

L: the planning cycle of a set of periodic tasks. It is computed as the least common multiple of $\{p_i : i = 1, 2, ..., N_T\}$.

$M_i \rightarrow M_j$: the precedence constraint imposed on modules M_i and M_j which means that the completion of M_i enables M_j to be ready for execution.

e_i: the execution time of a module M_i.

N_M: the number of modules to be allocated within a planning cycle.

K_{pn}: the number of PNs available for allocation.

x: an allocation which assigns modules to PNs. $x_{ik} = 1$ if module M_i is assigned to PN N_k.

TG: the task flow graph representing the task system.

$TG(x)$: the set of modules which are already allocated under the allocation x. $TG(x) = TG$ if x is a complete allocation.

AN: the set of active nodes in the search tree which can be considered for node expansion in the next stage.

x_0: a null allocation which corresponds to the root of the search tree.

x_{opt}: an optimal allocation.

P^*_{ND}: the objective function value achieved by x_{opt}.

$\hat{P}_{ND}(x)$: the value by which the objective function, P_{ND}, of all child nodes expanded from the allocation x is upper-bounded.

$P_{tc}(T_\ell \mid x)$: the probability that a task T_ℓ is completed before its deadline under allocation x.

TG_c: a component task graph of TG.

$\{TG_c\}$: the set of component task graphs of TG.

p_c: the probability that TG is represented by TG_c.

$TG_c(x)$: the set of modules $\in TG_c$ which are allocated under x.

$S_k(x)$: the set of modules which are assigned to N_k under allocation x, i.e., $S_k(x) = \{M_i : x_{ik} = 1\}$.

r_i: the release time of module M_i which can be interpreted as the earliest time at which M_i can start its execution.

LC_i: the latest completion time of M_i. M_i must be completed before LC_i to ensure all tasks to meet their deadlines.

D_i: the critical time of M_i. M_i must be completed before D_i to ensure the task to which M_i belongs to meet its deadlines.

C_i: the completion time of M_i which is determined by **MSA**.

$\hat{e}_i$: the *modified* execution time of M_i. $\hat{e}_i$ is used to include the effect of queueing M_i on the release times of all those modules that succeed M_i.

$f_i(C_i)$: the cost incurred by completing M_i at C_i.

$com_{ij}(x)$: the IMC communication time of M_i and M_j under allocation x.

d_{ij}: the IMC volume (measured in data units) between M_i and M_j.

$Y_{k\ell}$: the nominal delay (measured in time units per data unit) between two PNs, N_k and N_ℓ.

B: the minimal set of modules that are processed without any idle time from $r(B) = \min_{M_i \in B} r_i$ until $c(B) = r(B) + e(B)$, where $e(B) = \sum_{M_i \in B} e_i$.

b: the number of blocks in $S_k(x)$.

dg_i: the outdegree of M_i within a block under consideration.

$\hat{B}_i$: a subblock of $B - \{M_m\}$, where $1 \leq i \leq \hat{b}$, $\hat{b}$ is the number of subblocks in $B - \{M_m\}$, and M_m is the module scheduled to be executed if no other modules in B are waiting.

q_a: the looping-back probability of the loop L_a.

n_{L_a}: the maximum loop count of the loop L_a.

$q_{b,\ell}$: the branching probability of the ℓ-th branch of an OR-subgraph, O_b.

n_{O_b}: the number of branches in the OR-subgraph O_b.

$P_{tc}(T_\ell \mid TG_c, x)$: the probability that a task T_ℓ is completed before its deadline under x for a given component task graph TG_c.

$\hat{T}_\ell \triangleq \{M_i : M_i \in T_\ell \cap TG_c, dg_i = 0 \text{ with respect to } T_\ell \cap TG_c\}$: the set of modules without any successor in $T_\ell \cap TG_c$.

LP: the set of modules which are contained in loops.

OR: the set of modules which are on branches of OR–subgraphs.

λ_k: the *constant* exponential failure rate of N_k.

$\hat{\lambda}_{mn}$: the *constant* exponential failure rate of link ℓ_{mn}. We assume that λ_k's and $\hat{\lambda}_{mn}$'s are statistically independent of one another.

t_{mn}: the nominal delay (measured in time units per data unit) of link ℓ_{mn}.

$n(k,\ell)$: the number of edge–disjoint paths from N_k to N_ℓ.

$I(m,n,k,\ell)$: the indicator variable such that $I(m,n,k,\ell) = 1$ if ℓ_{mn} lies on one of the $n(k,\ell)$ edge–disjoint paths from N_k to N_ℓ.

$P_{IMC}(i,j,n_c)$: the probability that the IMC between M_i and M_j occurs n_c times in one task invocation.

$R_{mn}(i,j,n_c,x)$: the probability that link ℓ_{mn} is operational during n_c occurrences of IMC between M_i and M_j under allocation x.

$R_{\text{pn}}(x)$: the probability that all PNs are operational during the execution of modules assigned to them under allocation x.

$R_{\text{link}}(x)$: the probability that all links are operational for performing all the IMCs that use them under allocation x.

LC_i^p: a pessimistic estimate of LC_i used in the branching process.

LC_i^o: an optimistic estimate of LC_i used in the branching process.

r_i^o: an optimistic estimate of r_i used in the branching process.

$\widehat{PN}$: the set of PNs who need to reschedule the modules assigned to them because of the addition of $M_i \rightarrow N_k$ to a partial allocation.

Notation Used in Chapter 3

P_{dyn}: the probability of dynamic failure, or the probability of a task failing to complete before its deadline.

K_t: the number of state regions in region–change broadcasts.

TH_i, $1 \leq i \leq K_t - 1$: thresholds that divide the (workload) state space into K_t state regions.

e_i: the execution time of task T_i.

ℓ_i: the laxity of task T_i.

T_p: the probability update period for prior/posterior distributions.

Ω: the parameter space of a Bayesian decision problem.

W: the variable representing the outcome of the parameter space Ω.

O: the observation of the parameter W.

S: the sample space of all possible values of the observation, O.

D: the decision space of a Bayesian decision problem.

L: the loss function (defined on the product space $\Omega \times D$) of a Bayesian decision problem.

$\zeta(P,d)$: the expected risk for the given distribution, P, of the parameter W, and the decision d.

$\zeta^*(P)$: the Bayes risk defined as the greatest lower bound for the risks $\zeta(P,d)\ \forall d \in D$, i.e., $\zeta^*(P) = \inf_D \zeta(P,d)$.

$\{P_{O|W}(\cdot|\omega), \omega \in \Omega\}$: the family of sampling functions of observation O.

$P_{W|O=o}(\cdot)$: the conditional probability of the state, W, given the observation $O = o$.

$\delta(o)$: the Bayes decision function that specifies, for each possible observation $o \in S$, a decision $\in D$.

$\delta^*(o)$: the Bayes decision against the conditional distribution of W when $O = o$.

N_B: the number of nodes in a buddy set. For example, $N_B = 3m(m-1)+1$ if the buddy set is an H–mesh of dimension m.

w_i^a (w_i^b): the state of node i after (before) the change of state region in a region–change broadcast.

t_0: the time when a region–change broadcast is stamped.

a: the probability update ratio that represents the tradeoff between obtaining better averages and reflecting load changes.

λ^{ext}: external task arrival rate at a node.

$\{e_1, ..., e_k\}_{\{p_{e_1}, ..., p_{e_k}\}}$: the execution time distribution of external tasks, i.e., an external task requires e_i units of execution time with probability p_{e_i} in the task set.

$\{\ell_1, \ell_2, ..., \ell_n\}_{\{\hat{p}_{\ell_1}, \hat{p}_{\ell_2}, ..., \hat{p}_{\ell_n}\}}$: the laxity distribution of external tasks, i.e., an external task has ℓ_i time units of laxity with probability $\hat{p}_{\ell_i}$ in the task set.

$P_{dyn|\ell}$: the probability of missing the deadline of a task with laxity ℓ.

$P_{dyn|\ell,e}$: the probability of missing the deadline of a task with laxity ℓ and execution time e.

λ_{max}: maximum system utilization, i.e., the upper bound for λ below which $P_{dyn} \leq \epsilon$ can be achieved for some pre-specified $\epsilon > 0$.

r_{tt}: the task transfer–out ratio defined as the portion of arrived composite (both external and transferred–in) tasks that have to be transferred.

f_{tc}: the frequency of task collision defined as the fraction of transferred tasks that are not guaranteed on remote nodes after their transfer.

f_b, f_p, f_r: the frequency of region–change broadcasts, the frequency of state probing, and the frequency of request–for–bids, respectively.

C_b, C_p: the broadcast processing cost, and the probability–update processing cost.

CV: the coefficient of variation of the hyper-exponential task interarrival times used in the simulation.

Notation Used in Chapter 4

$\lambda(T)$: the composite (both external and transferred) task arrival rate at a node, given that the node's CET, T.

q_i ($1 \leq i \leq E_{max}$): the probability that an external (local) task requires i units of execution time.

$\hat{q}_j$ ($0 \leq j \leq L_{max}$): the probability that an external (local) task has laxity of j units of time.

B_k: the kth busy slot (measured in terms of system clock cycles) relative to any reference point of time.

T_k: a node's CET at the end of B_k.

X_k^i: the number of type–i task arrivals during B_k.

R: the number of clock cycles required for a node to complete a task which finds the node idle upon its arrival.

T^+,T: the CET on a node at some *embedding* time instant (i.e., at the end of each busy slot) and at some *random* time instant, respectively.

$p_T(\cdot)$, $p_T^+(\cdot)$: the distribution of T and the distribution of T^+, respectively.

$\alpha(T)$: the rate of transferring tasks out of a node given that the node's CET is T.

$\beta(T)$: the rate of transferring tasks *to* a node given that the node's CET is T.

γ_j: the probability that the CET on a node is no less than j units of time.

K_j: the number of nodes that can be chosen by a node, excluding the node itself, for transferring a task with laxity j.

$c_R(i)$, $c_P(i)$: the communication overheads encountered by a task with i units of execution time in the random selection scheme and the proposed scheme, respectively.

r_{tt}^R, r_{tt}^P: the task transfer–out ratio in the random selection scheme and the proposed scheme, respectively.

Notation Used in Chapter 5

$\{p_i(j), j = 1, ..., E_{max}\}$: the distribution of composite[1] task execution time on node i, where E_{max} is the maximum task execution time. This distribution will be estimated on–line by each node i.

$\{\hat{p}_i(j), j = 1, ..., L_{max}\}$: the distribution of composite task laxity on node i, where L_{max} is the maximum laxity. This distribution will also be estimated on–line by each node i.

$CET_i(\ell)$: the cumulative execution time (CET) on node i contributed by tasks with laxity $\leq \ell$ under the MLFS discipline.

$O_i(\ell)$: the observation about $CET_i(\ell)$ made by some node $j \neq i$.

$p_{C_\ell}(\cdot \mid O_i(\ell))$: the posterior distribution of $CET_i(\ell)$ given the observation $O_i(\ell)$. This posterior distribution is constructed by each node $j \neq i$ with the state samples collected via time–stamped region–change broadcasts.

$\overline{V}_{i,\ell}$: the event that future tighter–laxity task arrivals at node i do not invalidate the existing guarantee of a task with laxity ℓ.

$G_{i,\ell}$: the event that a task with laxity ℓ can be guaranteed by node i even in the presence of future tighter–laxity task arrivals.

λ_i: the exponential composite task arrival rate at node i.

$g_1(\lambda_i) = const$: the non-informative distribution which serves as the prior distribution in Bayesian estimation.

---

[1] both external and transferred

$f(t \mid \lambda_i) = \lambda_i e^{-\lambda_i t}$: the likelihood function of interarrival times given λ_i.

$f(\lambda_i \mid t_k)$: the posterior distribution of λ_i given the sample of interarrival time t_k.

$\mathcal{G}(\lambda \mid \alpha, \beta)$: the γ–distribution of λ_i with parameters α and β.

N_S: the size of statistical samples used to estimate λ_i, $\{\hat{p}_i(j)\}$, or $\{p_i(j)\}$.

$\boldsymbol{Y} = (Y_1, ..., Y_{L_{max}})$: the vector recording the numbers of laxity–j tasks in N_S task arrivals, where Y_j denotes the number of laxity-j tasks in N_S arrivals.

$\boldsymbol{p} : \boldsymbol{p} = (p_i(1), p_i(2), \ldots, p_i(L_{max}))$ is the vector of probabilistic parameters to be estimated.

$f(\boldsymbol{y} \mid N_S, \boldsymbol{p})$: the likelihood function of $\boldsymbol{Y}$ among N_S outcomes given $\boldsymbol{p}$.

$\mathcal{D}(\boldsymbol{p} \mid \boldsymbol{\alpha})$: the Dirichlet distribution of $\boldsymbol{p}$ with parameter $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_{L_{max}})$.

$F_k(t)$: the empirical distribution function of task interarrival times defined as the proportion of the observed samples which are $\leq t$.

$F_\lambda(t) = 1 - e^{-\lambda t}$: the hypothesized exponential distribution.

D_{ks}: the test statistic for the Kolmogorov–Smirnov test.

α_{ks}: the significance level used in the Kolmogorov–Smirnov test, i.e., the probability that the test falsely rejects the hypothesized distribution.

λ_i^{ext}: the external task arrival rate at node i.

$\overline{\lambda^{ext}}$: the average external task arrival rate. It is an index of system load.

K_n: the number of heterogeneous groups in the system.

r_n: the ratio of external task arrival rates between two 'adjacent' groups, i.e., $\frac{\lambda_{g+1}^{ext}}{\lambda_g^{ext}} = r_n$.

Notation Used in Chapter 6

CET_i: the cumulative task execution time (CET) on node i.

$\underline{T_Q} = (T_1; T_2; ...; T_{L_{max}})$: the description of the sorted task queue on a node, where $T_j \triangleq e_1^j e_2^j ... e_{j+1}^j$ is a record of tasks with laxity $j \in \{1, \ldots, L_{max}\}$ currently queued on a node, and $e_k^j \in \{0, \ldots, E_{max}\}$, $1 \leq k \leq j+1$, is the time required to execute the k–th laxity–j task in the queue.

$T_{out}^{<i>}$: the timeout period; node i will be diagnosed as failed if no broadcast message from node i has been received for this period since its latest broadcast.

λ_F: the exponential failure rate of a node.

$Ob(t)$: the indicator variable for the event that a broadcast message is received within time t.

T_{nb}: the random variable representing the time to node i's next broadcast (measured relative to the time of node i's last broadcast).

H_0 (H_1): the hypothesis that node i is operational (faulty).

π_0 (π_1): the unconditional probability that H_0 (H_1) is true.

p_0 (p_1): the probability density function of $Ob(t)$ under the hypothesis of H_0 (H_1).

$\delta(Ob(t)) \in \{0, 1\}$: the decision function of whether to accept H_0 or H_1 based on $Ob(t)$.

$P_F(\delta)$: the probability that H_0 is falsely rejected.

$P_M(\delta)$: the probability that H_1 is falsely rejected.

α_{ht}: the significance level of the hypothesis test used to determine the best timeout period.

$\{X(t), t \geq 0\}$: the continuous–time Markov chain on a finite state–space S, which models the state evolution of a node.

$Q = (q_{ij})$: the generator matrix of the continuous–time Markov chain $\{X(t), t \geq 0\}$, where q_{ij}, $0 \leq i, j \leq N$, is the transition rate from state i to state j and $|S| = N + 1$. Note that $q_{ii} = -\sum_{j=0, j \neq i} q_{ij} \stackrel{\Delta}{=} -q_i$, and $0 \leq q_i \leq \infty$.

K_{er}: the rate and the shape parameter K_{er} of the Erlang distribution which models the deterministic consumption of CET on node i.

$\underline{H} = (H_0; H_1; H_2; ...; H_{L_{max}})$: the state of a node, where $H_j \stackrel{\Delta}{=} h_1^j h_2^j ... h_{j+1}^j$ is a sequence of $j+1$ numbers with $h_k^j \in \{0, \ldots, K_{er} E_{max}\}$. h_k^j represents the number of service stages contributed by the k-th laxity-j task in the node queue.

$c_j \stackrel{\Delta}{=} \sum_{k=1}^{j+1} h_k^j$: the total number of service stages contributed by all laxity-j tasks.

$last(H_j)$: the index of the last nonzero entry in H_j, or, equivalently, the number of nonzero h_k^j's in H_j.

$L_{\mathbf{now}}(\underline{H})$: the laxity of the task currently in service.

$q_{\underline{H}, \underline{H}'_{\ell, K_{er} m}}$: the rate of the transition from $\underline{H}$ to $\underline{H}'_{\ell, K_{er} m}$ caused by queueing a newly-arrived task with ℓ time units of laxity and m units of execution time.

$q_{\underline{H}, \underline{H}'_{\ell, -1}}$: the rate of the transition from $\underline{H}$ to $\underline{H}'_{\ell, -1}$ caused by the consumption of 1 service stage of the task with laxity ℓ.

$\{Y_n, n = 0, 1, \ldots\}$: the discrete–time Markov chain abstracted from the continuous–time Markov chain $\{X(t), t \geq 0\}$ by randomization.

$\mathcal{P} = (\mathcal{P}_{ij})$: is the transition matrix of the discrete–time Markov chain $\{Y_n, n = 0, 1, \ldots\}$.

$\{N(t), t \geq 0\}$: the Poisson process abstracted from the continuous–time Markov chain $\{X(t), t \geq 0\}$ by randomization, such that $\{Y_{N(t)}, t \geq 0\}$ is probabilistically identical to $\{X(t), t \geq 0\}$.

Λ: the rate of the Poisson process $\{N(t), t \geq 0\}$.

S_j: the j-th state broadcast region. $S_j = \{\underline{H} : K_{er} \cdot TH_{2(j-1)} \leq \sum_{n=1}^{L_{max}} \{\sum_{k=1}^{n+1} h_k^n\} < K_{er} \cdot TH_{2j}\}$

$r_j(n, k)$, $0 \leq k \leq n + 1$: the probability that $\{Y_n\}$ visits the states in S_j k times out of n state changes.

$r_j(n, k, \underline{H})$: the probability that $\{Y_n\}$ stays in S_j k times out of n state changes and the state visited during the last transition is state $\underline{H}$.

N_n: the number of nodes in the distributed system, e.g., $N_n = 3e(e-1) + 1$ in H_e.

μ_F: the exponential node recovery rate.

$T_{fixed}^{<i>}$: a fixed timeout period used by node i in the simulation.

Notation Used in Chapter 7

H_e: A C–wrapped hexagonal mesh of dimension e.

m: the dimension of the buddy set which itself is a hexagonal mesh, i.e., the buddy set is a H_m.

d_i, $0 \leq i \leq 5$: the $(60 \cdot i)$-degree clockwise direction to the horizon.

L_i: the ith row in each direction in an hexagonal mesh.

k_i, $i = 0, 1, 2$: the number of hops from the source node to the destination node along the d_i direction. Negative value means that the move is along opposite direction, $d_{[i+3]_6}$.

$1/\mu_T$: mean task execution time.

$\{p_N(n), n \geq 0\}$: the probability density function of the queue length of a node.

$f_{D_i}(t)$: the probability density function of the time needed for a packet to travel i hops.

$f_{W_k}(t)$: the probability density function of waiting time for tasks with laxity k.

λ_{TT}: the rate of transferring tasks out of a node.

λ_{SC}: the rate of state–region–change broadcasts.

q_h: the probability of sending a task to a node h hops away.

R_i: the i-th state region.

T_{ii}: the mean recurrent time of R_i; the expected time until the first transition into R_i given that a node's state starts in R_i.

T_i: the average time the continuous-time Markov chain, $\mathcal{M}$, constructed in Section 7.3.1 spends in R_i.

π_k: the stationary probability that the underlying discrete time Markov chain for $\mathcal{M}$ stays in state k.

p_f: the probability that a packet arriving (and receiving services if necessary) at a node will be forwarded to one of its neighboring nodes.

Λ: the throughput rate of a node.

p_c: the probability of a packet cutting through an intermediate node.

$\overline{\ell_B}$: the mean length of broadcast packets.

$\overline{\ell_{TT}}$: the mean length of task transfer packets.

$\overline{\ell}$: the mean packet length.

λ_B, $\lambda_{A,B}$, $\lambda_{T,B}$: the rate of generating broadcast packets at a node, the rate of terminal broadcast packets arriving at a node, and the rate of transit broadcast packets arriving at a node, respectively.

λ_{TT}, $\lambda_{A,TT}$, $\lambda_{T,TT}$: the rate of generating task-transfer packets at a node, the rate of terminal task-transfer packets arriving at a node, and the rate of transit task-transfer packets arriving at a node, respectively.

ρ: the traffic intensity of the queueing network of interest.

r_{atio}: the distribution of task laxity is assumed to be geometric with $\hat{p_{\ell+1}} = r_{atio} \cdot \hat{p_\ell}$.

BIBLIOGRAPHY

[AAS86] M. Alam and U. M. Al-Saggaf. Quantitative reliability evaluation of repairable phased-mission systems using Markov approach. *IEEE Trans. on Reliability*, R-35(10):498-503, October 1986.

[AC88] Rafael Alonso and Luis L. Cova. Sharing jobs among independently owned processors. *IEEE Proc. 8th International Conf. on Distributed Computing Systems*, pages 282-288, 1988.

[AF89] Y. Artsy and R. Finkel. Designing a process migration facility: the Charlotte experience. *IEEE Computer*, 22(9):47-56, September 1989.

[Ber86] James O. Berger. *Statistical Decision Theory and Bayesian Analysis.* Springer-Verlag, 1986.

[BG87] Dimitri Bertsekas and Robert Gallager. *Data Networks.* Prentice-Hall, Inc., 1987.

[BLLK83] K. R. Baker, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Preemptive scheduling of a single machine to minimize maximum cost subject to release dates. *Operations Research*, pages 381-386, March-April 1983.

[BNG92] N. S. Bowen, C. N. Nikolaou, and A. Ghafoor. On the assignment problem of arbitrary process systems to heterogeneous distributed computer systems. *IEEE Trans. on Computers*, 41(3), March 1992.

[BS85] A. B. Barak and A. Shiloh. A distributed load-balancing policy for a multicomputer. *Software-Practice and Experience*, 15(9):901-913, 1985.

[BT83] Joseph A. Bannister and Kishor S. Trivedi. Task allocation in fault-tolerant distributed systems. *Acta Informatica*, 20:261-281, 1983.

[CA82] T. C. K. Chou and J. A. Abraham. Load balancing in distributed systems. *IEEE Trans. on Software Engineering*, SE-8(4):401-422, July 1982.

[CK87] Thomas L. Casavant and Jon G. Kuhl. Analysis of three dynamic distributed load-balancing strategies with varying global information requirements. *IEEE Proc. 7th International Conf. on Distributed Computing Systems*, pages 185-192, 1987.

[CL86] Hung-Yang Chang and Miron Livny. Distributed scheduling under deadline constraints:a comparison of sender-initiated and receiver-initiated approaches. *IEEE Real-time Systems Symposium*, pages 175-181, 1986.

[CL87] Wesley W. Chu and Kin K. Leung. Module replication and assignment for real-time distributed processing systems. *Proceedings of the IEEE*, 75(5):547–562, May 1987.

[CS87] Wesley W. Chu and Chi-Man Sit. A batch service scheduling algorithm with time-out for real-time distributed processing systems. *IEEE Proc. 7th International Conference on Distributed Computing Systems*, pages 250–257, 1987.

[CS91] Yi-Chieh Chang and Kang G. Shin. Load sharing in hypercube multicomputers in the presence of node failure. *Proc. of IEEE 21th International Symposium on Fault-Tolerant Computing*, pages 188–195, 1991.

[CSK90] Ming-Syan Chen, Kang G. Shin, and Dilip D. Kandlur. Addressing, routing, and broadcasting in hexagonal mesh multiprocessors. *IEEE Trans. on Computers*, 39(1):10–19, January 1990.

[CT83] M. L. Chaudhry and J. G. C. Templeton. *A First Course in Bulk Queues*, chapter 2–3, pages 58–61, 127–130. John Wiley & Sons, Inc., 1983.

[DeG70] Morris H. DeGroot. *Optimal Statistical Decisions.* Mcgraw–Hill, Inc., 1970.

[DeG86] Morris H. DeGroot. *Probability and Statistics.* Addison–Wesley Publishing Comp., second edition, 1986.

[DO91] F. Douglis and J. Ousterhout. Transparent process migration: design alternatives and the Sprite implementation. *Software – Practice and Experience*, 21(8):757–785, August 1991.

[DRS91] J. W. Dolter, P. Ramanathan, and Kang G. Shin. Performance analysis of message passing in HARTS: a hexagonal mesh multiprocessor. *IEEE Trans. on Computers*, C–40(6):669–680, June 1991.

[dSeSG86] Edmundo de Souza e Silva and H. Richard Gail. Calculating cumulative operational time distributions of repairable computer systems. *IEEE Trans. on Computers*, C–35(4):322–332, April 1986.

[dSeSG89] Edmundo de Souza e Silva and H. Richard Gail. Calculating availability and performability measures of repairable computer systems using randomization. *J. of ACM*, 36(1):171–193, January 1989.

[EB86] M. Engelhardt and L. J. Bain. On the mean time between failures for repairable systems. *IEEE Trans. on Reliability*, R–35(10):419–422, October 1986.

[ELZ86] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, SE–12(5):662–675, 1986.

[FB89] David Fernández-Baca. Allocating modules to processors in a distributed system. *IEEE Trans. on Software Engineering*, 15(11):1427–1436, November 1989.

[GH85] Donald Gross and Carl Harris. *Fundamentals of Queueing Theory.* John Wiley & Sons, Inc., second edition, 1985.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, N. Y., 1979.

[GM84] Donald Gross and Douglas R. Miller. The randomization technique as a modeling tool and solution procedure for transient markov processes. *Operations Research*, 32(2):343–361, March–April 1984.

[Gra77] W. K. Grassmann. Transient solutions in markovian queueing systems. *Comput. and Ops. Res.*, 4:47–53, 1977.

[HJ87] Anna Hać and Xiaowei Jin. Dynamic load balancing in a distributed system using a decentralized algorithm. *IEEE Proc. 7th International Conf. on Distributed Computing Systems*, pages 170–184, September 1987.

[HL86] Chi-Yin Huang Hsu and Jane W.-S. Liu. Dynamic load balancing algorithms in homogeneous distributed systems. *IEEE Proc. 6th International Conf. on Distributed Computing Systems*, pages 216–223, 1986.

[Hou90] Catherine E. Houstis. Module allocation of real-time applications for distributed systems. *IEEE Transactions on Software Engineering*, 16(7):699–709, July 1990.

[HS91] C.-J. Hou and K. G. Shin. Load sharing with consideration of future task arrivals in heterogeneous distributed real-time systems. *IEEE Proc. of 12th Real-Time System Symposium*, pages 94–103, December 1991. An enhanced version is accepted for publication in *IEEE Trans. on Computers*.

[HS92] C.-J. Hou and K. G. Shin. Module allocation with timing and precedence constraints in distributed real–time systems. *IEEE Proc. 13th Real–Time Systems Symposium*, December 1992. An enhanced version is accepted for publication in *IEEE Trans. on Computers* subject to minor revision.

[HS93a] C.-J. Hou and Kang G. Shin. Determination of an optimal retry time in multiple–module computing systems. *IEEE Proc. of 2nd Int'l Symp. on Uncertainty Modeling and Analysis*, pages 294–301, April 1993. An enhanced version is accepted for publication in *IEEE Trans. on Computers* subject to minor revision.

[HS93b] C.-J. Hou and Kang G. Shin. Incorporation of optimal timeouts into distributed real–time load sharing. *IEEE Proc. of 26th Hawaii Int'l Conf. on Systems Science*, January 1993. An enhanced version is accepted for publicaiton in *IEEE Trans. on Computers.*

[HS93c] C.-J. Hou and Kang G. Shin. Transparent load sharing in distributed systems: decentralized design alternatives based on condor package. submitted to *IEEE Proc. of 13th Int'l Conf. on Distributed Computing Systems*, 1993.

[HTT89] Jiawei Hong, Xiaonan Tan, and Don Towsley. A performance analysis of minimum laxity and earliest deadline scheduling in a real-time system. *IEEE Trans. on Computers*, C-38(12):1736–1744, December 1989.

[KC87] J. F. Kurose and R. Chipalkatti. Load sharing in soft real-time distributed computer systems. *IEEE Trans. on Computers*, C–36(8):993–999, August 1987.

[KF84] P. Krueger and R. Finkel. An adaptive load balancing algorithm for a multicomputer. Technical Report 539, University of Wisconsin–Madison, Dept. of Computer Science, April 1984.

[Kit88] John F. Kitchin. Practical Markov modeling for reliability analysis. *IEEE 1988 Proceedings Annual Reliability and Maintainability Symposium*, pages 290–296, 1988.

[KK79] P. Kermani and L. Kleinrock. Virtual cut–through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[KK93] L. Kleinrock and W. Korfhage. Collecting unused processing capacity: an analysis of transient distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 4(5):535–546, May 1993.

[Kle75] Leonard Kleinrock. *Queueing Systems, Volume 1: Theory.* John Wiley & Sons, Inc., 1975.

[KN74] L. Kleinrock and W. E. Naylor. On measures behavior of the ARPA network. *Spring Joint Comput. Conf., AFIPS Conf. Proc.*, pages 767–780, 1974.

[KS83] C. M. Krishna and Kang G. Shin. Performance measures for multiprocessor controllers. *Performance '83, A. K. Agrawala and S. K. Tripathi, Eds.*, pages 229–250, 1983. North-Holland.

[KS91a] Dilip D. Kandlur and Kang G. Shin. Design of a communication subsystem for HARTS. Technical Report CSE-TR-109-91, The University of Michigan, Dept. of Electr. Eng. and Comput. Science, Fall 1991.

[KS91b] Dilip D. Kandlur and Kang G. Shin. Reliable broadcast algorithms for HARTS. *ACM Trans. on Computer Systems*, 9(4):374–398, November 1991.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, pages 174–189, 1973.

[LL90] M. Litzkow and M. Livny. Experience with the Condor distributed batch systems. *Proc. of IEEE Workshop on Experimental Distributed Systems*, October 1990.

[LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor — a hunter of idle workstations. *Proc. of IEEE 8th Int'l Conf. on Distributed Computing Systems*, June 1988.

[LM82] M. Livny and M. Melman. Load balancing in homogeneous broadcast distributed systems. *Proc. ACM Comput. Network Performance Symp.*, pages 47–55, 1982.

[Lo88] Virginia Mary Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. on Computers*, C–37(11):1384–1397, November 1988.

[LT86] K. J. Lee and D. Towsley. A comparison of priority–based decentralized load balancing in distributed computer systems. *Proc. Performance '86*, pages 70–78, May 1986.

[MLT82] Perng-Yi Ma, Edward Y. S. Lee, and Masahiro Tsuchiya. A task allocation model for distributed computing systems. *IEEE Trans. on Computers*, C–31(1):41–47, January 1982.

[MTS89a] Ravi Mirchandaney, Don Towsley, and John A. Stankovic. Adaptive load sharing in heterogeneous systems. *IEEE Proc. 9th International Conf. on Distributed Computing Systems*, pages 298–306, 1989.

[MTS89b] Ravi Mirchandaney, Don Towsley, and John A. Stankovic. Analysis of the effect of delays on load sharing. *IEEE Trans. on Computers*, C–38(11):1513–1525, November 1989.

[MY84] Benjamin Melamed and Micha Yadin. Randomization procedures in the computation of cumulative-time distributions over discrete state markov processes. *Operations Research*, 32(4):926–944, July–August 1984.

[NH85] Lionel M. Ni and Kai Hwang. Optimal load balancing in a multiple processor system with many job classes. *IEEE Trans. on Software Engineering*, SE–11(5):491–496, 1985.

[NXG85] Lionel M. Ni, C. W. Xu, and T. B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Trans. on Software Engineering*, SE–11(10):1153–1161, 1985.

[OK92] O.Kremien and J. Kramer. Methodical analysis of adaptive load sharing algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 3(6):747–760, November 1992.

[Poo88] H. Vincent Poor. *An Introduction to Signal Detection and Estimation.* Dowden & Culver, Inc., 1988.

[PS87] Dartzen Peng and Kang G. Shin. Modeling of concurrent task execution in a distributed system for real-time control. *IEEE Trans. on Computers*, C-36(4):500–516, April 1987.

[PS89] Dar-Tzen Peng and Kang G. Shin. Static allocation of periodic tasks with precedence constraints in distributed real-time systems. *IEEE Proc. 9th International Conf. on Distributed Computing Systems*, pages 190–198, 1989.

[PTS88] Spiridon Pulidas, Don Towsley, and John A. Stankovic. Embedding gradient estimators in load balancing algorithms. *IEEE Proc. 8th International Conf. on Distributed Computing Systems*, pages 482–490, 1988.

[RKS90] P. Ramanathan, D. D. Kandlur, and K. G. Shin. Hardware assisted software clock synchronization for homogeneous distributed systems. *IEEE Trans. on Computers*, C–39(4):514–524, 1990.

[Ros70] Sheldon Ross. *Applied Probability Models with Optimization Applications.* Holden-Day, San Francisco, 1970.

[Ros83] Sheldon M. Ross. *Stochastic Processes.* John Wiley & Sons, 1983.

[RS88] P. Ramanathan and K. G. Shin. Reliable broadcast in hypercube multicomputers. *IEEE Trans. on Computers*, C–37(12):1654–1657, 1988.

[RSS90] Krithi Ramamritham, John A. Stankovic, and Perng-Fei Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Trans. on Parallel and Distributed Systems*, 1(2):184–194, April 1990.

[RSZ89] Krithi Ramamritham, John A. Stankovic, and Wei Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Trans. on Computers*, C–38(8):1110–1123, August 1989.

[SC89a] Kang G. Shin and Yi-Chieh Chang. Load sharing in distributed real-time systems with state change broadcasts. *IEEE Trans. on Computers*, C–38(8):1124–1142, August 1989.

[SC89b] Kang G. Shin and Yi-Chieh Chang. Load sharing in hypercube multicomputers for real-time applications. *4th Conf. on Hypercube, Concurrent Computers, and Applications*, pages 617–622, 1989.

[SC90] Kang G. Shin and Yi-Chieh Chang. A coordinated location policy for load sharing in hypercube multicomputers. submitted for publication, 1990.

[SH90] Kang G. Shin and C.-J. Hou. Analysis of three contention protocols in distributed real-time systems. *IEEE Proc. of 11th Real-Time Systems Symposium*, pages 136–145, December 1990.

[SH91] K. G. Shin and C.-J. Hou. Design and evaluation of effective load sharing in distributed real-time systems. *Proc. of Third IEEE Symposium on Parallel and Distributed Processing*, pages 670–677, December 1991. An enhanced version is accepted for publication in *IEEE Trans. on Parallel and Distributed Processing*.

[SH92] K. G. Shin and C.-J. Hou. Analytic models of adaptive load sharing schemes in distributed real-time systems. presented in Conf. of ORSA Computer Science and Operations Research: New Developments in Their Interfaces. An enhanced version is accepted for publication in *IEEE Trans. on Parallel and Distributed Systems*, 1992.

[SH93] Kang G. Shin and C.-J. Hou. Evaluation of load sharing in HARTS with consideration of its communication activities. *ACM 1993 Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, May 1993. An enhanced version has been submitted to *IEEE Trans. on Computers*.

[Shi91] Kang G. Shin. HARTS: A distributed real–time architecture. *IEEE Computer*, 24(5):25–34, May 1991.

[SKL85] Kang G. Shin, C. M. Krishna, and Y. H. Lee. A unified method for evaluating real-time computer controllers its application. *IEEE Trans. on Automatic Control*, AC–30:357–366, April 1985.

[SKS92] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load distributing for locally distributed systems. *IEEE Computer Magazine*, 25(12):33–44, 1992.

[SM89] J. K. Strosnider and T. E. Marchok. Responsive, deterministic IEEE 802.5 token ring scheduling. *Journal of Real-Time Systems*, 1(2):133–158, September 1989.

[Smi80] R. G. Smith. The contract net protocol: high-level communication and control in a distributed problem solver. *IEEE Trans. on Computers*, C–29(12):1104–1113, December 1980.

[SR88] John A. Stankovic and Krithi Ramamritham. *Tutorial Hard Real-Time System*, chapter 1, pages 1–11. IEEE Computer Society, 1988.

[SRC85] John A. Stankovic, Krithivasan Ramamritham, and Shengchang Chang. Evaluation of a flexible task scheduling algorithm for distributed hard real-systems. *IEEE Trans. on Computers*, C–34(12):1130–1141, December 1985.

[ST85] C. C. Shen and W. H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Trans. on Computers*, C–34(3):197–203, March 1985.

[Sta84] John A. Stankovic. Simulation of three adaptive, decentralized controlled, job scheduling algorithms. *Computer Networks*, 8:199–217, 1984.

[Sta85] John A. Stankovic. An application of Bayesian decision theory to decentralized control of job scheduling. *IEEE Trans. on Computers*, C–34(2):117–130, February 1985.

[Sto77] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. on software Eng.*, SE–3:85–93, January 1977.

[Stu88] M. Stumm. The design and implementation of a decentralized scheduling facility for a workstation cluster. *IEEE Proc. Secnod Conf. Computer Workstations*, pages 12–22, 1988.

[SW89] Sol M. Shatz and Jia-Ping Wang. Model and algorithms for reliability–oriented task–allocation in redundant distributed–computer systems. *IEEE Trans. on Reliability*, 38(1):16–27, April 1989.

[SWG92] S. M. Shatz, J.-P. Wang, and M. Goto. Task allocation for maximizing reliability of distributed computer systems. *IEEE Trans. on Computers*, 41(9):1156–1168, September 1992.

[TLC85] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the V–system. *Proc. of 10th Symp. on Operating System Principles*, December 1985.

[TT85] Asser N. Tantawi and Don Towsley. Optimal static load balancing in distributed computer systems. *Journal of the ACM*, 32(2):445–465, April 1985.

[TT89] Philip Thambidurai and Kishor S. Trivedi. Transient overloads in fault-tolerant real-time systems. *IEEE 10th Real-Time Systems Symposium*, pages 126–133, 1989.

[WM85] Yung-Terng Wang and Robert J. T. Morris. Load sharing in distributed systems. *IEEE Trans. on Computers*, C–34(3):204–217, March 1985.

[WM93] C. M. Woodside and G. G. Monforton. Fast allocation of processes in distributed and parallel systems. *IEEE Trans. on Parallel and Distributed Systems*, 4(2):164–174, February 1993.

[WS88] Abel Weinrib and Scott Shenker. Greed is not enough: Adaptive load sharing in large heterogeneous systems. *IEEE INFOCOM'88–The Conference on Computer Communications Proceedings*, pages 986–994, 1988.

[YL84] Takshing P. Yum and Hua-Chun Lin. Adaptive load balancing for parallel queues with traffic constraints. *IEEE Trans. on Communications*, COM–32(12):1339–1342, December 1984.

[YS81] Takshing P. Yum and Mischa Schwartz. The join-biased-queue rule and its application to routing in computer communication networks. *IEEE Trans. on Communications*, COM–29(4):505–511, April 1981.

[Zho88] Songnian Zhou. A trace–driven simulation study of dynamic load balancing. *IEEE Trans. on Software Engineering*, SE-14(9):1327–1341, September 1988.

[ZS92] Qin Zheng and K. G. Shin. On the ability of establishing real–time channels in point–to–point packet switched network. to appear in *IEEE Transactions on Communications*, 1992.