Realization of the Highly Integrated Distributed Real-Time Safety-Critical System MELODY

Dissertation

zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik

von

Jon Arthur Lind

Dortmund
1999

Tag der mündlichen Prüfung:	Januar 26,1999
Dekan/Dekanin:	Prof. Dr. Heinrich Müller
Gutachter:	Prof. Dr. Horst F. Wedde
	Prof. Dr. Heiko Krumm

Table of Contents

List of Figures

List of Algorithms

Acknowledgments

I would like to first thank my advisor Professor Dr. Horst F. Wedde. I feel indebted for the guidance and support he has extended to me over the years that we have known each other. He put an extraordinary amount of time and effort into developing and reviewing this thesis. Without his effort I would not have been able to write this thesis. His effort and dedication has meant more to me than I can say. Thank you very much!

I would also like to take this time to thank the other members of my committee, Professor Dr. Heiko Krumm, Professor Dr. Norbert Fuhr and Dr. Peter Buchholz. The time and effort spent reviewing this thesis is of great value to me.

To Bernd Klein, I am forever grateful for his invaluable time and effort in reviewing this thesis.

To all my colleagues at Lehrstuhl III, thank you all for the wonderful work environment and introduction to german culture and society. It has been a great experience that would not have been without you all.

To my wife, Heidi, I dedicate this thesis. For without her endless support and dedication throughout, along with her valued assistant in reviewing, this thesis would never have been completed. It’s a strong person that can endure through it all to see the end of this chapter in our life, and the beginning of the next. Thank you very much.

Additionally, I would like to thank my parents, Alfred and Norma, to whom this would not have been possible without some crucial guidance over the years. I also would like to thank all my family and relatives for the encouragement throughout the duration of my thesis.

Chapter 1 Introduction

1.1 Key Issues in Distributed Safety-Critical Real-Time Systems

Real-time systems are vital to a wide range of applications such as nuclear power plants, process control plants (automated factories), robotics, flight control (automatic piloting), control of automobile engines, laboratory experiments, Space Shuttle and strategic defense systems. Exactly what constitutes real-time computing has not yet been rigorously defined. Real-time systems cover the spectrum from the very small to the very large, and the size and *complexity* of such systems is growing. One of the main characteristics of real-time systems is that the timing constraints of real-time systems go beyond basic efficiency as characterized by notions of response times, throughput and utilization. In real-time systems, the time that an operation produces its results is as important (if not more important) to the success of the computation as is the logical correctness of the result. Normally this temporal correctness is defined in terms of deadlines. Thus, the basic difference between real-time systems and non-real-time systems is the notion of time. The traditional notion of fairness does not have the same significance in real-time computing. In some real-time systems it may not be possible for all tasks to meet their deadlines. In such real-time systems, when deadlines must be met, the criticality of tasks to the system's mission is more important than throughput and utilization. Critical tasks should be scheduled to meet their deadlines, even at the expense of less critical tasks. Predictability is a fundamental requirement of such systems.

This thesis focuses on a particular research effort for developing the novel distributed real-time operating system **MELODY**. MELODY has been specifically tailored to the requirements of *safety-critical* systems. A **safety-critical** system is a real-time system typically operating in an *unpredictable environment*. Examples of safety-critical system include: nuclear power plants, automated robot systems, automatic landing systems. Tasks in such systems not only have to meet their associated deadlines, but most of these are critical in the sense that the system would not survive in the case of a certain number of deadline failures of subsequent task instances. A task instance in such a critical stage would have a hard deadline, and is said to have become **essentially critical**. MELODY terms the ability to handle such essentially critical task instances a system's **survivability** [WeD91a]. Beyond this special real-time aspect of survivability, safety-critical system must also satisfy the following rigid **dependability** requirements: *reliability, fault tolerance and flexibility.* To satisfy these conflicting requirements, a very high degree of **adaptability** of system services is demanded. Such that, the safety-critical system is able to adjust **dynamically** to the changing environmental conditions typically found in safety-critical applications. Safety-critical systems have gained rapidly increasing relevance in research and development, both industrial and commercial, and are typically distributed. All this makes the design, implementation and testing of large safety-critical systems an extraordinarily complex modeling and engineering challenge.

Incremental Experimentation. Reliability and timing constraints (regarding external or environmental time) are conflicting requirements in distributed safety-critical real-time systems. To achieve reliability (and availability of information) multiple and mutually consistent copies of files are maintained. This is further enhanced by requiring that a minimum number of copies be available at any time. The real-time responsiveness of the system should be optimized by placing copies of a file at nodes were they are most frequently read while the system at the same time tries to reduce the number of copies that are frequently updated. Measures for achieving high reliability and availability are in conflict with those that improve the real-time responsiveness since a large number of mutually consistent file copies improves the availability of file information and the reliability of services depending on it while requiring a high communication overhead. This in turn reduces the chances that

writing task instances meet their deadlines. Under the assumption of unpredictable events in the application environment the trade-off between reliability and real-time responsiveness is dynamic. *Heuristic modeling and analysis procedures* are thus to be used, also due to the need for adaptivity in the system design. For taking into account the large number of relevant parameters, and their complex interdependencies, in such systems and at the same time designing the MELODY system in a transparent procedure we followed a novel incremental approach which we termed *incremental experimentation* [WeL97]. Incremental experimentation is a heuristic experimental, performance-driven and performance-based methodology that allows in an educated way to start with a coarse system model (with accurate logical expectations regarding its behavior). Through experimental investigation, these expectations are validated. If they are found to successfully stand the tests extended expectations and model features are generated for refining the previous design model (as well as its performance criteria). The refinement is done in such a way that the previous experimental configurations are extreme model cases or data profiles that both logically and experimentally are to reproduce the behavior of the previous modeling step. Thus novel performance aspects (or tendencies) could unambiguously be attributed to the influences of the refined model features. MELODY has currently progressed through five phases of incremental experimentation. The main basis for this thesis is the completion of this sixth phase in MELODY's development and distributed implementation. A technical presentation of the incremental progress in the MELODY project is to be found in [WeL97].

1.2 Direction and Goals of the Thesis

MELODY's development through five previous phases of incremental experimentation is outlined below:

Phase 1: The MELODY File System. In the first phase in the MELODY project, novel file system functions tailored to safety-critical requirements were designed [WeA90]. The functions were shaped to be both sensitive to real-time constraints and adaptable to the needs in the unpredictable environments in which safety-critical systems typically operate. This included the development of three system modules that interacted at each node: File Server, File Assigner, and Task Scheduler. The main objective was to develop algorithms that adaptively distribute (relocate, replicate and delete) file copies within the MELODY system.

Phase 2 & 3: Criticality and Sensitivity of Tasks. In practice task instances have a varying degree to which they are be sensitive to latest information (*this applies to read tasks only*). It is also reasonable that a number of subsequent deadline failures of task instances could be tolerated without endangering the whole system's survivability. The results from phase 1 provided basic expectations for refining the MELODY model by including criticality and sensitivity measures [WeX92]. Criticality was then defined as a relative measure based on the number of subsequent deadline failures and therefore its increasing impact on the system's survivability. **Sensitivity** was defined as the relative degree to which task instances needed access to latest file information. The concept is based on the number of subsequent deadline failures.

Phase 4: Distributed Resource Scheduling Algorithms. At the same time as phases 2 and 3 were being completed, novel distributed resource scheduling algorithms had been defined and validated in extensive comparative simulation experiments [Dan92, WeD91a, WeD91b and WeD94]:

Priority Insertion Protocol: This protocol was designed to minimize priority inversion. To avoid deadlocks, a call-back mechanism was included that would prevent lower priority tasks from keeping a lock while a higher priority task was already scheduled in a local priority scheduling queue.

Delayed Insertion Protocol:This protocol is designed to reduce the effects of cascaded blocking (that may take place between tasks that need mutually exclusive access to overlapping sets of shared resources). It is an extension of the Priority Insertion Protocol in which the inclusion into the scheduling queue is preceded by the tasks needing to achieve a *ready* status at all resources requested.

On-the-Fly Protocol:To eliminate cascaded blocking completely this protocol employs an *abort* (*suspend*) and *restart* strategy. A priority scheduler is used to determine the winning task at a resource (that has received a lock grant at all resources) while all other tasks which are

competing against the winner are determined to be losers. Losers will be suspended and placed into a local *sleep list* while being forgotten by all other resource managers. When a busy resource is released, the scheduler will look at its sleep list for candidates to be awakened and restarted.

When survivability (see section 1.1) of the algorithms was evaluated a key result was that the Priority Insertion and Delayed Insertion protocols performed significantly superior to the On-the-Fly protocol. Also, it had been found that for distributed safety-critical real-time systems, by the overall deadline failure rate performance (the standard performance measure in the real-time literature), the Delayed Insertion protocol performance was very good, even compared to the Priority Insertion protocol. Thus it was chosen to implement the distributed resource scheduling in MELODY. The Priority Insertion Protocol and the Delayed Insertion Protocol are the first examples of such algorithms, to our knowledge are the only ones, specifically designed and evaluated to be near optimal in distributed resource scheduling for distributed real-time safety-critical systems.

Phase 5: The Principle of Reversed Task and Resource Scheduling. In traditional operating system design, resource acquisition is done prior to task scheduling. A task instance that had achieved locks on all its required resources would then be scheduled/guaranteed by the Task Scheduler, or otherwise be subject to abortion. Throughout this entire time competing task instances would be blocked and may run out of time. This remote blocking would be *uncontrollable* by their local agencies. Distributed resource allocation occupies a comparably large portion of the task execution time, in contrast to the purely local task scheduling activities. Thus the abortion decision by the Task Scheduler would come very late. To instead abort tasks as early as possible and at the same time lock the resources as late as possible the principle was established [WeK93] to *reverse* the order of task and resource scheduling in MELODY. This leaves the Task Scheduler without accurate information on task execution times (the actual resource allocation time is unknown when the Task Scheduler is invoked) and a task instance would then have to be scheduled based on estimates. As a result the Task Scheduler can no longer guarantee a task instance will meet its deadline (since resource scheduling would be completed after task scheduling). However, by introducing a novel Run-Time Monitor module not only would the Task Scheduler abort task instances, but the Run-Time Monitor would supervise resource acquisition and would abort task instances as early as possible during their acquisition phase. This abortion would be an accurate decision since the execution time is known before the locking procedure begins. At the same time competing task instances would benefit from resources being locked as late as possible (after the task scheduling phase).

Integrated Task and Resource Scheduling. To satisfy the tight constraints found in safety-critical real-time systems it is necessary to control all activities that could effect a task instance's ability to meet its deadline. Since the purely local task scheduling activity takes much less time than (remote) file acquisition integration of task and resource scheduling means most likely to invoke the Task Scheduler during the resource scheduling activities of the File Server. Integration of task scheduling activities would have to handle the following conflicting goals:

- Infrequent invocation of the Task Scheduler. This minimizes the overhead caused by Task Scheduler activities (increasing the time for resource scheduling activities) while at the same time the number of task instances waiting to be scheduled is increased thus providing for *wiser* scheduling decisions. This policy equally means to invoke the Task Scheduler *as late as possible*.
- Invocation of the Task Scheduler *as early as possible*. This increases, for every individual task instance, the time available for acquiring the needed resources.

To adaptively handle these issues a dynamic integration policy was developed [WeL94 and WeL97] which would invoke the Task Scheduler based upon the level of competition and the number of task instances waiting to be scheduled.

Upon completion of Phase 5, ***simulated*** results regarding MELODY's file system features had been obtained. Initial simulation regarding the integration of task and resource scheduling had been conducted, however, the insights were limited to investigating task profiles where survivability was not at stake. At this point no work had been done to also integrate the File Assigner and Run-Time Monitor activities. This is a major objective of this thesis. Finally, the current step (**phase 6**) in the MELODY

system project constituted a major model extension, from *simulation* to ***distributed experiments***, that explicitly reflects actual task computation, file manipulation (read and write operations) and real communication traffic. In detail, the need for distributed experiments will be described next.

Simulation and Distributed Experimentation. For the purpose of the MELODY simulations a number of simplifications had been made to the system model. The computation time of a task instance had been modeled to be selected from a given interval of time. The task instances were then simulated by placing each of them on an execution queue and generating an event at a certain time for its completion. File manipulation was handled in such a way that the read/write request was placed onto an access list with a completion event being generated during a prespecified time interval in the future. Communication in the simulators was increasingly refined. The initial simulators (phase 1 through 4) had only generated a message to be sent, and they placed it into the receiving node's communication queue. This message was then handled during the next simulation time unit thus suggesting that the communication took just one (simulation) time unit. The simulator in phase 5 refined the simulation of the communication medium such that the message transmission time was selected from a bounded interval of time. In all simulations, the assumption was made that messages were neither lost nor duplicated, and were received in the order sent. Virtual task computation, file manipulation, and communication were done to simplify the implementation of the simulator.

However, the effect that communication, real task computation and file manipulation would have as compared to the simulation model, was an open question. In particular, since the distributed model had been implemented on a single processor it was not possible to realistically implement distributed task computation and file manipulation, or to study the real overhead caused by invoking the corresponding system services. Also, actual communication within a distributed real-time environment is designed to have a strict upper bound. As the load on the real communication medium increases (due to unpredictable task arrival rates, file update frequency or file transfers) so does the potential for the communication time to exceed any designed bounds. The load on the communication medium also effects the basic assumption that no message is lost and that the order of messages is preserved, like in case of the UDP protocol which is used by our current distributed implementation, for synchronization messages. This makes communication a key issue for realistic experimental studies.

This thesis addresses the mentioned issues. Taking this into account it was hoped that by modifying and varying model parameters, task and data profiles the simulation set-ups and assumptions could be modeled as extreme circumstances in the distributed set-ups. At the same time it was expected that the results/tendencies of the simulation studies could be recovered under the corresponding extreme distributed set-ups while they would be refinable or extendible to the general distributed experimental situation. This is the line of discussion in the subsequent chapters, and will be explained there in detail.

The major focus of the distributed experiments is on the performance of MELODY's integration policies. The performance criteria employed are the global deadline failure rate (see the paragraph behind **Phase 4**) and survivability (see section 1.1). In the experiments the following combined measure was used: the deadline failure rate was averaged for a given number of experimental runs, however if a run failed to survive then the outcome of the experimentation was counted as (deadly) failure at this one observed failure point since survivability is understood as a worst-case measure.

To summarize, the goals for the thesis are:

1: Develop and apply the novel incremental experimentation methodology to implement MELODY as an academic prototype of a distributed real-time safety-critical system.

2: Specifically report on the particular performance studies regarding the step-wise *conceptualization, development, and integration* of the MELODY servers (File Server, Task Scheduler, File Assigner and Run-Time Monitor).

1.3 Outline of the Thesis

In chapter 2, previous and related work will be discussed. The technical concepts and features of the MELODY system are introduced and summarized in chapter 3. The technical model and implementation of each local MELODY server is described in detail in chapter 4 (File Server), chapter 5 (Task Scheduler), chapter 6 (File Assigner), and chapter 7 (Run-Time Monitor). In chapter 8, initial experimental results are discussed that were used to verify the implementation of the distributed implementation of MELODY. The following chapters discuss experimental results (including experimental set-ups and expectations) regarding the unique and novel integration policies implemented in MELODY. The experiments on the integration models used for the Task Scheduler/File Server integration are discussed in chapter 9. In chapter 10, experiments on the integration policies for the File Assigner are reported. In chapter 11, the integration of the Run-Time Monitor is reported. Final conclusions are given in chapter 12. Specific information regarding the communication model (appendix A) and time synchronization protocol (appendix B) is supplied. C-language source code for the distributed implementation of MELODY is not included in this thesis (directions for obtaining it along with the distributed experimental set-ups can be found in appendix C).

Chapter 2 Previous and Related Work

The MELODY project was started at Wayne State University in Detroit, Michigan, evolving from an extensive Ph.D. research study [Ali88]. For MELODY's development through five previous phases of incremental experimentation see section 1.2. These previous stages of the MELODY project have been reported in [Dan92, WeA90, WeD91a, WeD91b, WeX92, WeD94, WeK93, WeL94, WeL97, and WeL98]. Although valuable related work (detailed below) has been reported on studying issues regarding: run-time monitoring, task scheduling, and file assignment and data similarity such discussions have never been fully integrated into a comprehensive view. Likewise these approaches do not take into account the unpredictability in safety-critical environments. Thus adaptivity has not been discussed.

Run-Time Monitoring. Despite extensive work on monitoring and debugging facilities for distributed and parallel systems, run-time monitoring of real-time systems has received little attention (with a few exceptions). Special hardware monitoring for collecting run-time data in real-time applications has been considered in a number of papers [CJD91, MoL97, RRJ92 and TKM88]. These approaches introduce specialized co-processors for the collection and analysis of run-time information. The use of special-purpose hardware allows for non-intrusive monitoring of a system by recording the run-time information in a large repository, often for post-analysis. In related work [HaS89, TFC90 and HaW90] the use of monitoring information to aid in scheduling tasks has been studied. Here the under-utilization of a CPU due to the use of scheduling methods based on the worst-case execution times of tasks is approached through a hardware real-time monitor that measures the task execution times and delays due to resource sharing. The monitored information is fed back to the system for achieving an adaptive behavior.

Previous work has been reported on studying on-line run-time monitoring of timing constraints [OhM96]. In this model, a system computation is viewed as a sequence of event occurrences. The system properties that must be maintained are then expressed as invariant relationships between various events, which are monitored at run-time. If a violation of an invariant is detected, the system is notified so that suitable recovery options can be taken. The invariants are specified using a notation based on real-time logic, and timing constraints are allowed to span processors. The run-time monitoring facility monitors and detects violations in a distributed fashion. This allows violations of the local machine to be detected as early as possible. The monitor consists of a set of cooperating monitor processes (at each processor). Application tasks on a processor inform the local daemon of events as they occur. The daemon on the local machine checks for violations as local events happen while sending information about event occurrences needed by remote machines to other remote daemons.

Other related work has been reported on the application of real-time monitoring for scheduling tasks with random execution times [HaS90]. The main focus of this work is to feed back monitored information to the system for an *a posteriori* improvement of the behavior. Specifically, the analyzed results about task execution behavior are funneled back to the host's operating system and used for the dynamic scheduling of tasks. The monitor provides accurate and timely information about task execution behavior. The real-time monitor is then used to measure the elapsed pure execution time, which is used on-line to calculate the anticipated future execution time. The comparison between the worst case execution time and the pure execution time is used to determine an estimated execution time. This estimated execution time is used to determine at an earliest possible time if the task can meet its deadline.

In MELODY, the Run-Time Monitor is responsible for using run-time monitor information maintained by the File Server to determine, at the earliest possible time, when to abort the task instance. The Run-Time Monitor uses actual values from prior instances of a task to determine an estimated execution time and estimates for the time required to complete acquisition of shared resources. Using these estimates the Run-Time Monitor is then able to determine the earliest point of time at which it can check the current task instance (based on its current deadline). At this earliest point of time the Run-Time Monitor uses also the estimates for the remaining time required for the task to determine whether to abort the task instance. This results in a run-time monitoring facility that is adaptive to changing environmental conditions that would effect the estimated times.

Task Scheduling. Tasks and their schedulability has been studied [ABD95, HLF95, LiL73, NaS94, RSL89 and XuP93] in regards to the potential advantages of off-line task scheduling for hard real-time systems. Off-line scheduling, however, is not reasonable in safety-critical real-time systems due to the inherent unpredictability of the system environment. An off-line scheduling algorithm would have to take into account the unpredictability of the tasks and adjust the schedule to handle this unpredictability. This unpredictability in the scheduling would then have a significant effect on the quality of the schedule and the performance of the operating system. On-line scheduling is therefore typically the standard method of task scheduling in safety-critical real-time systems from which work for example has been reported in [LeM80 and LeW82]. In on-line scheduling, the schedule for tasks is computed as tasks arrive while the scheduler is not assumed to have any knowledge about the major characteristics of tasks that have not yet arrived. The major advantage of this approach is its adaptability to handle unpredictable environments. However, there are two fundamental disadvantages with on-line scheduling in safety-critical real-time systems. The first problem is the limited ability of the task scheduler to guarantee that the timing constraints will be satisfied, because there is always the possibility that a newly arrived task will make a previously scheduled task miss its deadline. The second problem is the amount of time available for the scheduler to compute schedules on-line is severely restricted.

A well-understood on-line scheduling algorithm for guaranteeing the deadlines of periodic tasks in a distributed real-time system is the rate monotonic scheduling algorithm [LiL73]. Under this algorithm, fixed priorities are assigned to tasks based upon the rate of their requests (tasks with relatively shorts periods are given a relatively high priority). However, the scheduling problem for aperiodic tasks is very different from that for periodic tasks. Scheduling algorithms for aperiodic task must be able to guarantee the deadlines for periodic tasks while providing good average response times for aperiodic tasks even though the occurrence of the aperiodic requests are non-deterministic. Two common approaches for servicing these aperiodic tasks are *background processing* and *polling* [LSS87 and SLR86]. Background servicing of aperiodic tasks would occur whenever the processor is idle. If the load of the periodic tasks is high, then the utilization left for background service is low, and background service opportunities are relatively infrequent. The polling method consists of creating a periodic task for servicing aperiodic tasks. At regular intervals, the polling task is invoked and services any pending aperiodic requests. However, if no aperiodic tasks are pending, the polling server suspends itself until its next period and the time originally allocated for aperiodic tasks is not preserved but is instead used by periodic tasks. Even though polling and background processing can provide time for servicing aperiodic tasks, they have the drawback that the average response times for the algorithms are unpredictable (especially for background processing). In safety-critical systems the relatively long average response time would have a significant impact on the system's ability to schedule essentially critical task instances having a severe impact the system's survivability.

The *sporadic server* algorithm [SLS88 and SSL89] has been developed to provide faster average response times for aperiodic tasks while guaranteeing the deadlines of periodic tasks. As with polling, the sporadic server algorithm creates a high priority periodic task for servicing aperiodic tasks. The sporadic server accommodates aperiodic tasks by inserting them into a free slots in the "periodic schedule" (which is likely to happen as long as aperiodic tasks occur sporadically). However unlike polling, the sporadic server yields improved average response times for aperiodic tasks because of its

ability to provide immediate service for aperiodic tasks (due to it's high priority). *Slack stealing* is a relatively new approach which offers improvements in average response times over the sporadic server [ShG92 and SpB96]. The Slack stealing algorithm is an optimal service method for the sporadic server based on the idea of "stealing" all the possible processing time from the periodic tasks. The amount of slack time available during a given interval is equal to the amount of time that can be used to execute aperiodic requests without causing any periodic request to miss its deadline.

In MELODY, due to the unpredictable environment which is typical for safety-critical applications, aperiodic tasks are not assumed to arrive sporadically (as in safety-critical environments) thus the two techniques discussed cannot be applied. Rather, in MELODY the order of task and resource scheduling has been reversed (see section 1.2) with invocation of the Task Scheduler occurring during the resource scheduling activities of the File Server. Invocation of the Task Scheduler is then based upon a dynamic method (rather than a static priority) which takes into account both the number of task waiting to be scheduled and the current number of competing task instances in the File Server.

File Assignment Problem. In distributed computing determining the optimal number of file copies and their distribution is known as the file assignment problem. Improper distribution of files is likely to yield a poor performance. This poor performance is the result of overhead on the operating system caused by remote access, and the additional communication required to access a remote file copy also adds transmission costs to the overhead. While read access to a low number of file copies would cause access delays (since it is more likely that a task will need to access a remote copy) it improves write performance (due to fewer number of mutually consistent copies needing to be updated). This inherently conflicting issue in turn then effect issues such as file availability and system reliability. The file assignment problem directly addresses how to minimize the local node and global network costs of the system by distributing the correct number of file copies among a set of nodes. The local node costs are decomposed into the storage cost and the CPU cost for processing the information such as updating, retrieving, and storing. Network cost are decomposed into communication and transmission costs. This problem of determining a solution to this problem under varying system conditions has been addressed in a number of publications [ChL87, DoF82, MoL77, Mur83, NoA87, Pol94, Pu86, RoS96, SeS79, Seg76, Yu85 and ZSA92].

Solutions to the file assignment problem can be divided into two classes: static file assignment and dynamic file assignment. In static file assignment [MoL77, Mur83, and Pu86], the design of the file assigner is based upon a set of time-invariant system parameters. Therefore, in solutions based on static file assignment the distribution of files is determined prior to system start-up and their locations remain unchanged for the entire operation period. Clearly in an unpredictable environment (such as assumed for MELODY), static file allocation is totally unrealistic and has limited applications. Dynamic file assignment [LeM78, SeS79, Seg76, Smi81, and Yu85] is defined in terms of time-variant file location and system parameters such as the file request rate and file access patterns. In these approaches the file access patterns are monitored, and then a static file assignment function is invoked whenever the access patterns change. In MELODY, the File Assigner is responsible for adaptively changing the distribution of file copies according to the changing and typically unpredictable environment [Ali88, Sta96 and WeL97]. Rather than invoking the File Assigner based upon changes in the file access patterns, MELODY invokes the File Assigner based on changing task request patterns and deadline failure rates. The File Assigners then utilizes information monitored by the local and remote sites to determine which of the many alternatives should be taken to improve the survivability and deadline failure rate performance of the local site.

Imprecise Computations & Similarity. The value of a data object that models an entity in the real world cannot be updated continuously to perfectly track the dynamics of the real-world entity. The time required to perform the update alone introduces a time delay which means that the value of a data object can not instantaneously be the same as the real-world entity. In critical system situations it may well be more important for the survivability to rely on data values which are not quite up-to-date yet timely available, than to wait for accurate values and potentially miss a critical deadline. Weaker

correctness criteria can be introduced for allowing transactions to utilize two different but similar interchangeable values without adverse effects. In particular, the data values of a data object that are slightly out-of-date (but locally available) are often usable as read data for a task. Two data objects would be viewed as similar if the corresponding values in the data objects are viewed as similar.

Recently work has been done exploiting similarity between task instances to improve the performance of real-time data-intensive applications [HKM97, KaR95, KuM93, and RaP95]. In a schedule, two task instances are considered similar if they are of the same type and access similar values of the data object. An algorithm to produce a reduced schedule has been proposed [HKM97] to exploit the trade-off between data consistency and system work-load. This reduced schedule is generated based upon the observation that certain similar consecutive task instances are insignificant to the system. The real-time scheduler would then skip unimportant task instances with the hope of satisfying timing constraints and/or safety requirements within the system. This algorithm further reduces the schedule by removing one of two similar task instances in a schedule which would still preserve similarity of the output, and in this way data consistency within the system would not be violated. In MELODY tasks are assumed to be critical in a varying degree to which the survival of the system depends on their timely completion. If a task instance misses its deadline it is assumed that it would be more critical that the subsequent task instance meets its deadline. At some time, following a number of deadline failures, the task instance would become essentially critical and must meet its deadline (or else the system may not survive) (see section 1.1). In this way MELODY views the previous failed instances of a task as insignificant in respect to the system's survivability.

The imprecise computation technique is a way to provide flexibility in scheduling and resource management [LLL92]. For many applications, one is willing to accept the results of a poorer quality when the desired results of the best quality cannot be obtained. This is especially true for safety-critical applications where every essentially-critical task must meet its timing constraints. The imprecise computation technique improves the chances that a critical task will meet its deadline by making sure that an approximate result of an acceptable quality is available to the task when the result of the desired quality cannot be obtained. One method to avoid timing faults is to leave less important tasks unfinished if necessary [LiS90 and LNL87]. In this way, rather than treating all tasks equally, the system views important tasks as mandatory and less important tasks as optional. This is then further extended such that every task is composed of two subtasks: a mandatory subtask and an optional subtask. The imprecise computation technique makes meeting deadlines in real-time systems significantly easier for the following reason. To guarantee that all timing constraints are met, the system needs only to guarantee that all mandatory subtasks are completed, while optional subtasks can be completed as time permits. In MELODY weakly consistent local file copies (Private copies) are meant to be dynamically provided in due time at a site for enhancing the chances that an essentially critical task instance will be able to meet its deadline, and in turn preserve the system's survivability. This enhanced performance is due to faster local access times in comparison to remote access times which would incur additional communication overhead. More generally, tasks in MELODY are viewed as sensitive relative to the degree in which they need to utilize latest file information. With an increased number of subsequent deadline failures, the need to obtain up-to-date file information is increasingly neglected such that locally available information would be utilized for the sake of meeting the deadline and ensuring that the system survives (see section 1.2).

Chapter 3 MELODY Model

The model of a distributed safety-critical application is shown in figure 3-1. Here, there is a safety-critical environment in which the application and operating system have been distributed. Typically, safety-critical systems designate that certain nodes are providing specific services to the safety-critical application (due to specialized hardware and software located at these nodes). The model of such a safety-critical system also allows that certain nodes only provide system services to remote application nodes, but contain no application tasks (node C in figure 3-1). The safety-critical application in this environment is then held responsible for: defining which tasks are located at which nodes; and determining when task instances are created, and associating deadlines with those instances. Due to the unpredictable environment the time between two instances of the same task cannot be predetermined by the application.

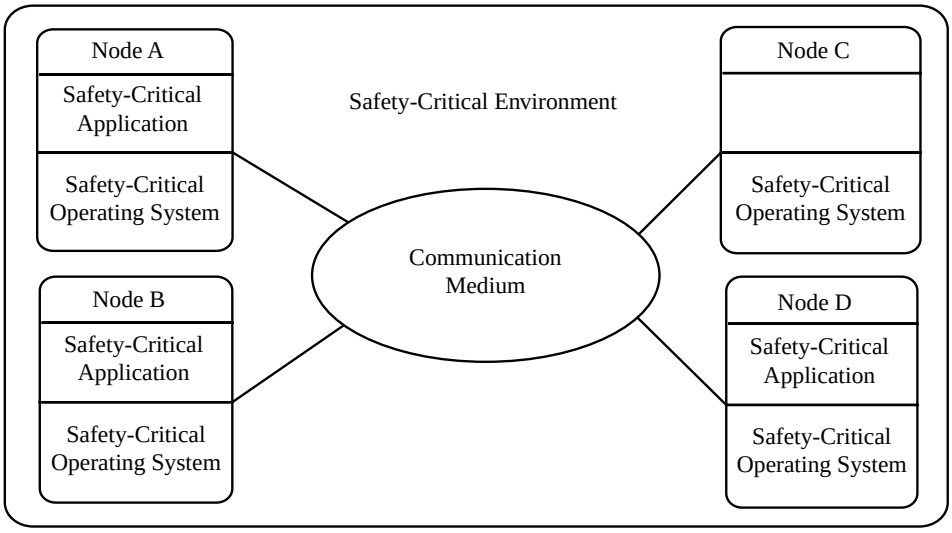


Figure 3-1: Safety-Critical Environment

In the example of an Automated Landing System (ALS) (as mentioned in section 1.1), *tasks* would be operations such as contributing to trajectory adjustments for approach and landing. *Task instance*s would have to be completed under increasingly rigid and increasingly critical time constraints. An ALS would have to take into account the following conditions when generating task instances and associating their deadlines:

- Allow for subsequent adjustments that would altogether guarantee the correct angle with respect to the runway under unpredictably changing wind conditions (e.g. gusty wind);
- The delay of the aircraft's reaction (approximately 4 sec. for a 747) makes a correction after a certain point of time obsolete. If the course is still wrong at this time the aircraft may crash onto the runway.

If a guidance system is to be used to determine the position with respect to the runway, then the deadlines of some task instances could possibly be missed. This being dependent upon the amount of “unexpected” information in the guidance system under processing. The closer a critical point of time comes, after which a collision with the ground becomes unavoidable, the tighter are those deadlines of corrective task instances. In this sense these tasks, and their instances, become increasingly critical, and eventually *essentially critical* (see also the formal concepts in section 3.1). The last instance of a task with a deadline before the *point of no return* (essentially critical task instance) must succeed. Otherwise, the airplane (and thus the ALS inside) would be endangered (possibly destroyed).

3.1 Task Model

Tasks represent control functions, corrective actions etc. They are executed on a regular basis. Every occurrence of a task is termed a task instance or incarnation. In MELODY, due to the unpredictability of the environment typical in most safety-critical applications, they are assumed to be aperiodic in nature. They may also be executing on dedicated processors. File access is done using remote file operations rather than task migration or file transfer to the accessing site. Each task is a small-scale, transaction-like operation with just one segment of task activity termed *critical section* in which it accesses a number of objects (copies of possibly different files) concurrently. Write operations work on all copies of a file, through remote local write operations, while read operations read from only one copy. We also assume that task execution times on each local file copy could be determined within tight bounds.

3.1.1 Criticality and relative degrees of Criticality

Tasks in safety-critical systems are critical in a varying degree to which the survival of the system depends on their timely completion. We will define a formal concept of task criticality in the following way:

Definition 1: For each site **I**, site-dependent integer thresholds $\mathbf{a}_i$' and $\mathbf{a}_i$" (which could be readjusted e.g. under changing hardware technology) will represent a *frame of criticality* (figure 3-2) for all tasks arriving at **I**. Each task T_j at site **I** is then assigned an individual *criticality value* C_j. If $C_j \geq \mathbf{a}_i$", then T_j is considered as a *non-critical* task, if $C_j \leq \mathbf{a}_i$', then the task is an *essentially critical* task. If $\mathbf{a}_i' < C_j < \mathbf{a}_i$", the task is called *critical*.

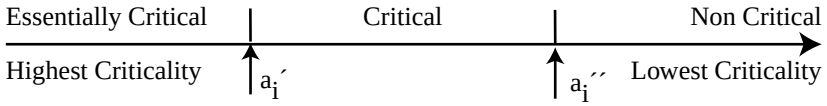


Figure 3-2: Frame of Criticality

C_j is understood then to be the criticality of the initial instance of task T_j, and would typically be experimentally determined (tailored to the application). Then, the essential step is to define a *relative degree of criticality*:

Definition 2: Let C_j be the criticality value of T_j, and T_{jk} the k^{th} instance or incarnation of T_j. The *relative criticality of instance* T_{jk} is the integer C_{jk} defined as follows:
$C_{jk} := C_j$; $C_j \leq a_i'$ or $C_j \geq a_i$" or $T_{j(k-1)}$ met its deadline;
otherwise
$C_{jk} := \max\{(C_j$- # of failures after last successful instance completion), $a_i'\}$

3.1.2 Sensitivity and relative degrees of Sensitivity

In a similar way read tasks need latest information to a varying degree. With an increased number of subsequent instance failures, however, the need of obtaining latest file information is more and more neglected, for the sake of making the deadline. Similar to definition 1 we define a formal concept of task sensitivity:

Definition 3: For each site **I**, site-dependent integer thresholds $\mathbf{b}_i$' and $\mathbf{b}_i$" will represent a *frame of sensitivity* (figure 3-3) for all tasks arriving at **I**. Each task T_j at site **I** is then assigned an individual *sensitivity value* R_j. If $R_j \geq \mathbf{b}_i$", then task T_j is called *robust*. If $R_j \leq \mathbf{b}_i$', the task is considered as *essentially sensitive*. If $\mathbf{b}_i' < R_j < \mathbf{b}_i$", the task is called *sensitive*.

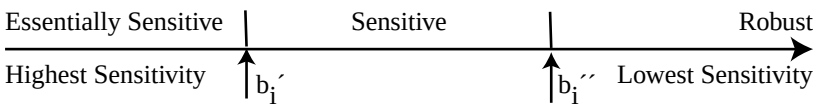


Figure 3-3: Frame of Sensitivity

R_j is understood to be the sensitivity value of the initial instance of task T_j, and would typically be experimentally determined (tailored to the application). Then, the essential step is to define *relative degrees of sensitivity*:

Definition 4: Let R_j be the sensitivity value of T_j, and T_{jk} the k^{th} instance of T_j. The *relative sensitivity of instance* T_{jk} is an integer R_{jk} defined as follows:
$R_{jk} := R_j$; $R_j \leq b_i$' or $R_j \geq b_i$" or $T_{j(k-1)}$ met its deadline;
otherwise
$R_{jk} := \min\{(R_j$ + # of failures after last successful instance completion), b_i"$\}$

Thus the relative criticality (relative sensitivity) of the instances of T_j is constant if T_j is essentially critical or non-critical (essentially sensitive or robust). Also, once the threshold values a_i' or b_i" have been reached by a task instance T_{jk} the value C_{jk} does not change, except after successful completion of T_{jk}. Finally the instance T_{jk} of task T_j is called *critical*, *non-critical*, *essentially critical* (*sensitive*, *robust*, *essentially sensitive*) depending how C_{jk} (R_{jk}) relates to the threshold values a_i', a_i" (b_i', b_i") as given above.

Tasks such as described in the example, discussed in the beginning of this chapter, typically need up-to-date information since the environment as depicted may change quickly and broadly, and therefore they are sensitive according to definition 3. If public copy information would come too late for the next task instance to complete within its deadline while waiting for the analysis of the latest information to be reflected in the *public* copies and if the next task instance were essentially critical, it is reasonable for the survival of the system to make use instead of local *private* copies, even care for having such copies available when they are needed. In this way there would be a *chance* to avoid the loss of the system. Correspondingly, with subsequent task instances failing the next instance is considered less sensitive, or more robust. In table 3-1 and table 3-2 a few task histories should further clarify the concepts.

Instance #k	1	2	3	4
Status	Failed	Failed	Success	
Criticality $\mathbf{C_{1k}}$	4	3	2	4
Sensitivity $\mathbf{R_{1k}}$	4	5	6	4

Table 3-1: Sample Failure History for Task T_1 (a_i' := 2 =: b_i', a_i" := 8 =: b_i")

In table 3-1, task T_1 has C_1=R_1=4 as initial criticality and sensitivity values, respectively. Instance T_{11} fails to meet its deadline. T_{11}'s failure results in instance T_{12} having a decreased relative criticality C_{12}=3 (T_{12} is more critical than T_{11}), and an incremented relative sensitivity R_{12}=5 (i.e. T_{12} is more robust than T_{11}). The failure of instance T_{12} again implies changes of T_{13}'s relative criticality and sensitivity values to 2 and 6, respectively. T_{13} has become essentially critical but completes by its deadline. The success of T_{13} causes T_{14}'s relative criticality and sensitivity values to be reset to the initial values of C_1 and R_1, respectively. However, had instance T_{13} not been successful the entire system might no longer have survived since it was essentially critical.

Instance #k	1	2	3	4
Status	Failed	Failed	Success	
Criticality $\mathbf{C_{2k}}$	6	5	4	6
Sensitivity $\mathbf{R_{2k}}$	7	8	8	7

Table 3-2: Sample Failure History for Task T_2 (a_i' := 2 =: b_i', a_i" := 8 =: b_i")

A failure history for task T_2 with an initial criticality C_2=6 and sensitivity R_2=7 is shown in table 3-2. After the failure of T_{21}, instance T_{22} is robust. As also T_{22} fails the relative sensitivity of T_{23} remains at 8 since T_{22} was already robust. Instance T_{23} is able to complete successfully, so T_{24}'s relative criticality and sensitivity values will be set to the initial values C_2 and R_2, respectively.

3.2 File Model

Concurrency control. Concurrency control in the sense of data consistency among a number of replicated data objects has been broadly discussed in distributed databases (recently also including real-time transactions, see chapter 2). Strong concurrency control protocols offer error-free data access, but require longer response times due to synchronization delays. Weak concurrency control protocols

reduce the communication overhead, but provide less error-free data access, and might not always be sufficient for a system. To provide adaptive real-time services in a safety-critical environment, MELODY maintains two types of file copies: **public** and **private**. Public copies utilize a strong concurrency protocol to guarantee that they are kept mutually consistent. Private copies utilize only a weak concurrency protocol (insuring that the copy is refreshed some time after the public copy is updated). For every public copy their exist a **shadow** copy from which tasks would read even while the corresponding public copy is being updated. The shadow copy thus provides for concurrent read/write access to public copies (this technique has been used frequently for non-replicated files, and has been extensively described in the literature (see chapter 2)). Write operations always work on all copies of a file, while read operations would be accommodated by either a shadow or private file copy, the latter being locally available. Whether an instance would tolerate using information from a private copy depends on its criticality and sensitivity (see section 3.1).

Public copies of a file are shared resources within the distributed real-time operating system MELODY. Write operations are performed on all public copies by utilizing a strong concurrency protocol. Write operations are multicasted (see appendix A) to all sites requiring information regarding the update. The strong concurrency protocol used in MELODY causes the access times for write operations to depend directly on the number of file copies rather than on their location. This results from additional overhead caused by an increasing number of public copies, while changes in public copy locations do not necessarily increase the overhead. In MELODY updating of a public copy is controlled by the local File Servers (see chapter 4) utilizing the *Delayed Insertion* protocol (see section 4.1.1). This protocol provides a method for logically locking all copies simultaneously prior to execution of the update operation. The updating of a *shadow copy* is a locally atomic action in conjunction with the public copy update operation. This atomic action guarantees that no new write operation is allowed to be performed until the completion of the update to the shadow copy. However, read operations are allowed to continue using either an *old* version of the shadow copy (which is the latest copy while the public copy is being updated) or the *new* shadow copy (while the *old* shadow copy is being transformed into the *new* public copy). To further enhance the reliability of public copies a minimum number of public copies should be available at any time (for information of crucial relevance). In this way ensuring that information would be available in the case of site or link failures.

An example of a public copy update of file F at a site I can be seen in table 3-3. At time 0, a public copy update is requested by task instance T_{21}, and will not complete updating the public copy until time 15. At time 10, a read operation is requested by task instance T_{11}, is started on the shadow copy and will continue until time 20. At time 15, the updating of the public copy is completed. The File Server then designates this copy as the *new* shadow copy, and as a result all future read operations will be performed on this copy. The File Server then proceeds to transform the old shadow copy into the new public copy version. This is done by applying the update to the old shadow copy before responding to task instance T_{21} that the update has been completed. However, task instance T_{11} is still currently reading the old shadow copy, therefore the updating of the public copy must wait until the completion of task instance T_{11}. The consistency of the information read by task instance T_{11} is still accurate since the update operation on the public copy has not yet been completed. Therefore, the information read is still the most current (readable) information available at this site. At time 20, the read operation for task T_{11} is completed. Following the completion of task instance T_{11}, the File Server detects that there are no longer any outstanding read requests for the old shadow copy. At this time it proceeds to perform the update operation (from task instance T_{21}) on the old shadow copy. At time 25, task instance T_{31} requests to read the shadow copy of file F. Its request is granted and the read operation proceeds on the new shadow copy. At time 35, the update operation is completed on the old shadow copy and it's then designated as the public copy version, and task instance T_{21} is informed that the update has completed. Whether the update was successful would be based on whether the update operation was completed prior to the deadline at all sites holding a public copy. Also at time 35, the read operation for task instance T_{31} is completed.

Time	Task Instance	Operation		File F (copy a)	File F (copy b)	Operation		Task Instance
0	T_{21}	write	-->	Public	Shadow			
5								
10					Shadow	<--	read	T_{11}
15	T_{21}	completed	<--	Public New Shadow	Old Shadow			
20					Old Shadow	--> <--	completed update	T_{11} T_{21}
25	T_{31}	read	-->	New Shadow				
30								
35	T_{31}	completed	<--	Shadow	Old Shadow Public	-->	completed	T_{21}

Table 3-3: Public Copy Update Example

Private copies of a file are a non-sharable resource among sites within the MELODY system, while they provide weakly consistent information to local robust or essentially critical reading task instances. They are requested when the robust (see section 3.1) read access rate to a file is high at a certain site and the updates are rather infrequent. MELODY would also request a private copy when it determines that the next instance of a task T_j would be essentially critical (see section 3.1). Maintaining a private copy of the file at a site would enhance the chance for specific task instances to meet their deadlines under all circumstances when necessary. This enhanced performance results from faster access times for local access, in comparison to remote access (with its inherent communication overhead). Therefore, as the number of private copies of a file increases the expected time required to access a copy of this file is reduced. The local File Server ensures that a private copy is refreshed after a public copy update operation has occurred. A reading task instance would use a local private copy if T_{jk} is considered *essentially critical* or *robust*. Otherwise, the read operation must be performed on a shadow copy. The characteristics of whether the task is robust and how fast it becomes essentially critical are aspects that would be tailored by the application using MELODY's parameter settings (see section 3.1).

3.3 Task Life Cycle

A task instance T_{jk} goes through the following four phases to successfully complete before its deadline (figure 3-4):

Scheduling: Instance T_{jk} is placed into an execution schedule by the Task Scheduler.
Location: The list of required files for task instance T_{jk} are located by the File Server.
Acquisition: All necessary file copies for instance T_{jk} are acquired by the File Server.
Computation: The instance performs its read/write operation on the acquired files in cooperation with FS.

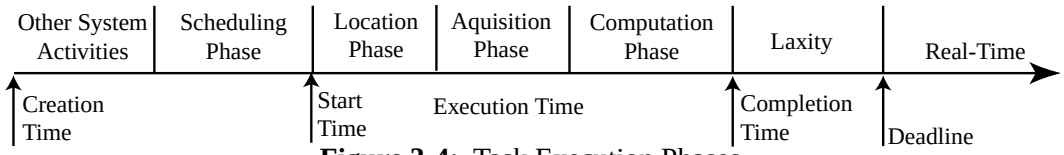


Figure 3-4: Task Execution Phases

An application creates a task instance T_{jk} at a specific time (determined by the application in regards to the real-time environment) that will be termed as the *creation time* of the task instance. Following its creation, task instance T_{jk} is sent, by the application, to MELODY together with its relative degrees of criticality and sensitivity (which are based on the prior task instance $T_{j(k-1)}$). If instance T_{jk} is essentially critical, it will be sent to the local File Server (section 3.4) to begin its location phase. Otherwise, it is sent to the local Task Scheduler (section 3.4) to begin its scheduling phase.

During the **scheduling phase** of a task instance T_{jk}, the Task Scheduler tries to schedule the task instance based on its scheduling algorithm (see chapter 5). Once the instance has been scheduled it is sent to the File Server to begin its location phase. However, if the task instance is determined to be non-schedulable, it is aborted and removed from the system. More detailed information regarding the task scheduling in MELODY can be found in chapter 5.

Task instances received by the File Server are to begin their associated **location Phase**. The File Server locates all files in the list of required files (LRF_{jk}) for an instance T_{jk}. Writing task instances locate all public file copies. Reading task instances locate either a local copy (shadow or private based on the task instance's relative degrees of criticality and sensitivity) or a remote shadow copy (private copies can not be accessed remotely). An instance completes its location phase and begins its acquisition phase after all files have been located. At this point the time required for the computation phase of the task instance is known based on: the number of files accessed; physical location of the file copies; and the operation to be performed (read/write).

A task instance that is in the **acquisition phase** acquires files based on the type of access required. Writing task instances go through the two sub-phases to complete their acquisition phase: **allocation** and **locking**. These two sub-phases for writing task correspond directly to the steps 1 through 3 in the *Delayed Insertion Protocol* (see section 4.1.1). In the **allocation phase** a task instance sends *access* request messages to all file copies and waits to acquire *ready* messages from copies of all files in it's LRF_{jk}. Only after acquiring all *ready* messages will the instance start the locking phase. During the **locking phase** the instance sends *schedule* messages to all copies of all files in it's LRF_{jk} and waits to be granted locks from the file copies. A writing task instance completes the acquisition phase only after being granted locks from copies of all files in it's LRF_{jk}. Reading task instances are not required to compete for access to either shadow or private file copies. However, they are required to be queued at the site holding the copy and receive in response an *acquired* message (read lock) before beginning their computation phase. Once a writing (reading) task instance has received a *ready* (*acquired*) messages from a File Server holding a copy of the file, the file copy will be guaranteed not to be physically deleted until the task instance has either completed its computation phase or is aborted and releases the copy.

If during the location and acquisition phases of a task instance it is no longer possible for the task instance to complete before its associated deadline it will be aborted, and all files acquired are released before the task is removed from the system. A task instance that has completed its acquisition phase can no longer be aborted. Whether the task instance completes successfully before its deadline is determined by the File Server.

An instance that has completed its acquisition phase begins its **computation phase** based on the order determined by the Task Scheduler. Once the task instance has begun its computation phase it uses a remote procedure call (RPC) mechanism to send the read/write operation to all files. The task instance then waits to receive a response back from all files. A response with a *successful* status is sent back to the task instance if the File Server (at the location where the file copy resides) was able to perform the operation (read/write) on the file before the task instance's associated deadline. Otherwise, a response with a *failed* status is returned. The determination as to whether a reading task instance T_{jk} has completed before its associated deadline, is made when the response from the read operation is received by T_{jk}. If the response is received prior to deadline DT_{jk} then T_{jk} is determined to be successful, otherwise T_{jk} has failed (this is reflected in the relative degree of criticality and sensitivity of the next task instance of $T_{j(k+1)}$). The File Server, however, guarantees that once a writing task instance T_{jk} broadcasts its write operation to all sites, involved in the write operation, the write will be completed regardless of whether or not the operation has been completed prior to deadline DT_{jk}. This ensures consistency of the MELODY file copies since some sites may have already performed the write operation prior to the deadline expiring (MELODY also provides no mechanism to handle a roll-back on the task level in order to avoid the high overhead this would create). The determination as to whether a writing task instance T_{jk} has completed prior to its deadline, is then based on the status of response to the write operation returned by each site. If any site responded with a *failed* status then T_{jk} is marked as having failed.

3.4 System Model

In MELODY four major modules interact at each node: File Server, Task Scheduler, Run-Time Monitor and File Assigner (figure 3-5). The first two are standard operating system functions while the

File Assigner and Run-Time Monitor realize quite novel services in MELODY. All modules are realized through local agents interacting under distributed control. Each module is briefly described below (specifications can be found in the following chapters).

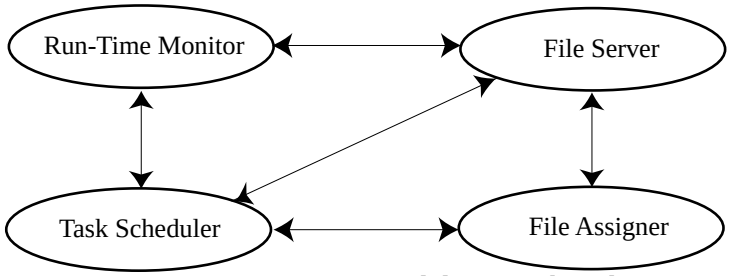


Figure 3-5: MELODY Modules at each node

File Server (**FS**). Each File Server engages in allocating concurrently the needed files for some k tasks. These are selected from the Competing Task Queue (CT). In our present distributed implementation the allocation is done through one of our novel distributed resource scheduling algorithms, the *Delayed Insertion Protocol* (see section 4.1.1), which is near-optimal both for minimization of deadline failures and survivability [Dan92]. Upon learning about an update of a file, i.e. of its public copies, the File Server, at a site holding a private copy, requests to refresh the private copy using a remote procedure call from a site holding a public copy. The File Server also maintains a read/write access history for each file requested by a task. This is used to calculate an estimated execution time (EET_{jk}) for a task instance T_{jk} to the set of needed files (LRF_{jk}):

$$EET_{jk} = \text{Estimated\_Location\_Time}(T_j) + \text{Estimated\_Acquisition\_Time}(T_j) + \sum_{r \in LRF_{jk}} \text{Computation\_Time}(r) \quad (3.4\text{-}1)$$

The Estimated_Location_Time and Estimated_Acquisition_Time for T_j is determined as the average of the corresponding values of the last 5 instances of T_j. The Computation_Time is the time required to access the needed copies of file r, including the communication time for the possibly involved remote operations.

Task Scheduler (**TS**). The Task Scheduler schedules a set of all tasks that have arrived at a site I according to a scheduling policy tailored to the application needs. A task instance T_{jk} arriving at site I (*unpredictably*), would be considered for scheduling using the following parameters:

EET_{jk}: Estimated Execution Time of a task instance T_{jk};
DT_{jk}: Deadline for task instance T_{jk};
LRF_{jk}: List of required files for T_{jk}, containing for every file its identifier and access pattern;
PrT_j: Static Priority.

Upon arrival of T_{jk}, the local Task Scheduler communicates with the local File Server to check which of the files in LRF_{jk} are locally available, and calculates an estimate for the actual execution time based on EET_{jk}. Tasks are then placed into the local task queue (LTQ) according to their order of execution, and dispatched for execution from this queue. *Due to the unpredictability of the environment, and the ensuing typical aperiodicity of task instance occurrences, task scheduling in MELODY is non-preemptive*. The Task Scheduler also characterizes tasks as either: **Strongly Schedulable** (T_{jk} can be inserted into LTQ and meet its deadline), or **Weakly Schedulable** (T_{jk} is decided to fail but could have met its deadline in case that all files from LRF_{jk} had been locally available, or the number of public copies would have been smaller).

File Assigner (**FA**). An essential adaptive feature in MELODY is its handling of distributed information resulting in strongly consistent (public) file copies, distributed over different system sites for both better availability and reliability. Their number, location, and quality are adaptively rearranged to ensure improved availability and system survivability under hard real-time constraints. This adaptive control of file copy distribution is handled by the novel File Assigner module. The File Assigner must make a trade-off, under changing request and deadline failure patterns, between the costs of serving file requests with a given distribution of public copies, and the costs for realizing various alternative distributions. The term “costs” here denotes time delays for overhead operations, communication and

transmission delays, both for remote and local communication. A large number of public copies is advantageous for sensitive read tasks since a copy is the more likely to be locally available. However, write tasks at the same time suffer from this since the cost for updating grows with the number of copies (as discussed previously). Each local File Assigner then cooperates with the File Server and with remote File Assigners to manage the replication, relocation, and deletion of public file copies within the MELODY file system. Private copies are not managed by the File Assigner since decisions for creation or deletion are solely local and require no consensus from remote sites.

Run-Time Monitor (**RTM**). As discussed previously (see section 1.2), to abort tasks as early as possible and at the same time lock the resources as late as possible the principle was established to *reverse* the order of task and resource scheduling in MELODY. This leaves the Task Scheduler without accurate information on task execution times (the actual resource allocation time is unknown when the Task Scheduler is invoked) and a task instance would then have to be scheduled based on estimates. As a result the Task Scheduler can no longer guarantee a task instance will meet its deadline (since resource scheduling would be completed after task scheduling). However, by introducing a novel Run-Time Monitor module not only would the Task Scheduler abort task instances, but the Run-Time Monitor would supervise resource acquisition and abort task instances as early as possible during their acquisition phase. This abortion would be an accurate decision since the execution time is known before the locking procedure begins. At the same time competing task instances would benefit from resources being locked as late as possible.

Also discussed in section 1.2, to satisfy the tight constraints found in safety-critical real-time systems it is necessary to control all activities that could effect a task instance's ability to meet its deadline. Since the purely local task scheduling activity takes much less time than (remote) file acquisition, integration of task and resource scheduling means most likely to invoke the Task Scheduler during the resource scheduling activities of the File Server. Integration of task scheduling activities would have to handle the following conflicting goals:

- Infrequent invocation of the Task Scheduler. This minimizes the overhead caused by Task Scheduler activities (increasing the time for resource scheduling activities) and the context switching overhead while at the same time the number of task instances waiting to be scheduled is increased thus providing for 'wiser' scheduling decisions (selection from a larger set of waiting task instances). This policy equally means to invoke the Task Scheduler *as late as possible*.
- Invocation of the Task Scheduler *as early as possible*. This increases, for every individual task instance, the time available for acquiring the needed resources.

To adaptively handle these issues a dynamic integration policy under the control of the Run-Time Monitor was developed to invoke the Task Scheduler based upon the level of competition and the number of task instances waiting to be scheduled.

Chapter 4 File Server

In MELODY, local File Server (FS) modules are responsible for handling all aspects regarding access to physical file copies located at the site. The File Server also handles the competition of local task instance's for local/remote file copies. This includes the following services:

- Complete requests, from local task instances, for the allocation of files (local/remote);
- Allocate local copies of files to task instances (located either local/remote);
- Handle physical access (read and write) to local copies of files;
- Maintain file history information for all files accessed by local tasks, or for file copies at the site;
- Maintain task history information for all tasks located at the site;

4.1 File Server Model

Each local File Server works in cooperation with remote File Servers to complete requests to files that may or may not be locally available. The File Server module uses a client/server approach to handle the management of local task instances and local file copies. The division of responsibilities between the File Server client and server has been subdivided as follows:

Client: Responsible for handling activities regarding competing task instances located at a site;

Server: Responsible for managing all activities regarding access to copies of files located at a site using the *Delayed Insertion protocol* (details follow).

This separation of the File Server client and server responsibilities is shown in figure 4-1. As a result of the separation of responsibilities, the FS client has no direct access to file copies located at the site. The FS client must however communicate (internally) with the local FS server to gain access to those copies.

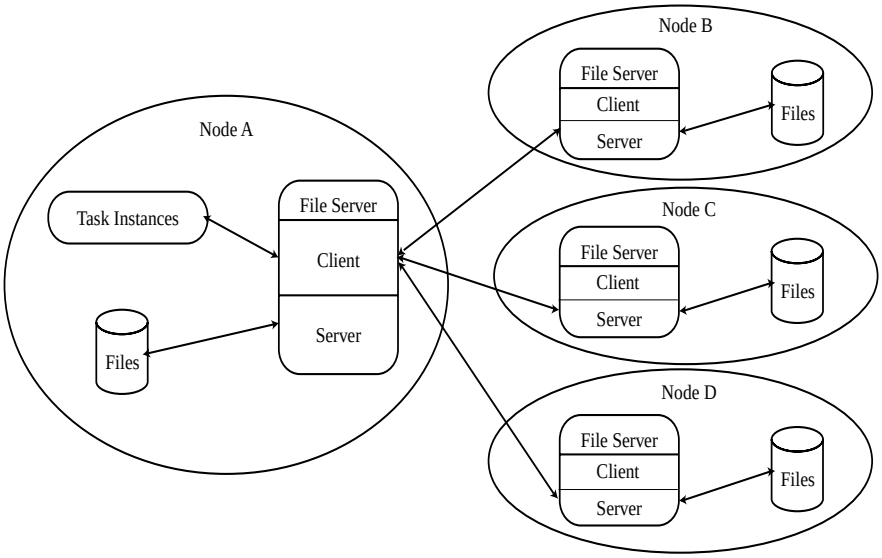


Figure 4-1: File Server Client/Server Model

4.1.1 Delayed Insertion Protocol

The Delayed Insertion protocol is a distributed resource scheduling algorithm that has been defined and validated in extensive comparative simulation experiments by D.C. Daniels [Dan92, WeD91a, WeD91b, and WeD94]. This protocol is designed to reduce the effects of *cascaded blocking* that may take place between tasks that need mutually exclusive access to an overlapping set of shared resources. Cascaded blocking results in poor utilization of the resources due to the fact that tasks must

wait for other tasks to complete before accessing their set of resources. As a result of the loss of concurrency the average task turnaround times are longer due to longer blocking durations. Though the protocol does not eliminate cascaded blocking, it employs a heuristic designed to reduce its occurrence. When a task requests to access some *busy* file, the file manager places the task in a wait list until the file is released and becomes free. When released and no other task is scheduled for access at the file, the task is then notified about a *ready* status by the file manager. If T_j has received the *ready* messages about all files ($r \in LRF_j$) then T_j requests to be scheduled in the execution queue at all files ($r \in LRF_j$). The following is a short description of the delayed insertion protocol. A detailed explanation of the protocol and its properties (along with proof for the protocol being starvation free, deadlock free and priority inversion is limited to the blocking duration of one executing task) can be found in [Dan92].

A file manager M_i maintains an execution schedule E_i which is ordered by priority. Each task instance that arrives for resource scheduling will have an associated priority. The schedules maintained by the file manager will be order by the lexicographical ordering among relative criticality, deadline, task instance id and host id. The procedure INSERT(t,s), which inserts a task instance into the schedule according to the lexicographical order just mentioned. Let C_{jk} and DT_{jk} be the parameters associated with task instance T_{jk} respectively. If two task instances are found in any two schedules, they will be found in the same order in those schedules. Provided that the task at the front of the schedule always holds the lock for a resource, the Delayed Insertion protocol is deadlock free. The lock for a resource is always granted to the task instance at the front of the scheduler. However, there is no guarantee that at any given time the task instance at the front of the schedule holds the lock for the resource. Moreover, this ambiguity can lead to disagreement between the schedulers which may lead to deadlock. To resolve this condition, we introduce the **Call Back Mechanism**.

From time to time, in order to avoid deadlock, it may be necessary for a scheduler to request that a lock which has been granted to a task be released by that task. This situation will occur whenever a scheduler inserts a task into a non-empty schedule such that the task instance is at the front of the schedule. Locks are always granted to the task which is at the front of a non-empty schedule. As the ordering between messages sent from different sites is not preserved, it is possible for access requests from different task instances to arrive at the front of a schedule in different orders. Because it is the policy of a scheduler to grant access to the task at the front of its schedule, a disagreement (potentially deadlocking) between schedulers over which task should proceed is possible. The call back mechanism then solves this problem be utilizing the three rules defined below. Schedulers follow rules 1 and 3, while task follow rule 2.

Rule 1: If some task instance holds the lock, then when the scheduler inserts a task instance into a non-empty schedule, such that the newly inserted task instance is at the front of the schedule and no *call back* message has been sent to the task instance holding the lock, then the scheduler will send a *call back* message to the task instance holding the lock (on behalf of new higher priority task instance).

Rule 2: If the task instance is considered executing, then upon receipt of a *call back* message from a resource, the task instance will ignore the message. Otherwise the task instance will temporarily release the lock by responding with a *call back acknowledge* message and will remove the previously granted lock from its list of granted locks. The task does not release the other locks which it holds.

Rule 3: Upon receipt of a *call back acknowledge message*, the receiving scheduler grants the lock to the instance at the front of the schedule (sending a *grant* message to the instance).

Note that when a call back is ignored, the task which holds the lock (the one that ignored the call back) will continue executing and release the lock upon completion. A call back acknowledge message is treated the same as a release message by the schedulers, except that a release message implies that the sending task is no longer an active client of the scheduler. When a client releases, the schedulers discard all information pertaining to that it. Servers maintain information only on their current, active clients.

The heuristic behind scheduling is that a newly arriving task is held out of the schedule of executing tasks until the server's backlog of currently scheduled tasks have been serviced. Thus the

delayed tasks cannot block other tasks at remote sites while they wait locally. Cascaded blocking can only occur between tasks that have been inserted into the schedule. In addition to the schedule E_i, a file manager M_i maintains two additional queues. Let W_i represent the set of *waiting* tasks at file manager M_i and let C_i represent the set of *candidate* tasks at file manager M_i. Taken together, W_i and C_i represent a secondary schedule in which tasks wait until they are ready at all needed files. When the execution schedule for file F_i is emptied ($M_i = \emptyset$), all tasks in the wait queue are promoted to the candidate queue and are sent ready messages. When a task has been informed that it is a candidate at all files ($r \in LRF_j$) then it requests to be placed in the execution schedule at all files ($r \in LRF_j$) by sending the needed resources a schedule message. Let FR_t be the set of files that have sent ready messages to task T_j, and let FL_t be the set of files that have sent lock grant messages to task T_j. The following is a description of the operations performed on both the Task (client) and File Manager (server) sides.

4.1.1.1 Task (client) Operations

Task operations are sequential for the delayed insertion protocol and defined by the following steps:

Step 1: As task instance T_j begins the acquisition phase at some site $I \in M$. The C_j (criticality) and D_j (deadline) values of the T_j are known ($FL_j = \emptyset$, $FR_j = \emptyset$). An access request (including the values of SP_j and D_j) is sent to all file managers ($F \in LRF_j$). GOTO step 2.

Step 2: Task instance T_j will wait for all needed files to respond with a ready message. As file F_i responds it is added to the set of ready files, FR_j ($FR_j = FR_j + F_i$). If ($FR_j = LRF_j$), T_j requests to be scheduled in the execution queue at resources $r \in LRF_j$, and GOTO step 3.

Step 3: Task instance T_j now waits for all files to respond with lock grant messages. As each lock is granted, the sending file F_i, is added to set FL_j ($FL_j = FL_j + F_i$). If ($FL_j = LRF_j$),T_j will GOTO step 4. Note that only during this step will a task instance acknowledge a call back message (they are ignored in all other steps). This acknowledgment is done by releasing the lock (previously granted) and then waiting again to receive the lock grant message from the site.

Step 4: Task T_j will begin execution.

Step 5: Upon completion, T_j will be released from all files ($F \in LRF_j$) and enter its inactive phase.

Note that when a task is aborted due to a failure to meet its deadline it sends *release* messages to all resources.

4.1.1.2 File Manager (server) Operations

File manager operations for the delayed insertion protocol are completely message driven. The behavior of the file manager can be expressed as message handlers that define the actions taken in response to incoming messages. A File Manager (M_i) then responds to access, schedule, call-back and release request messages and execute the corresponding handler *access*, *schedule*, *call-back* or *release*. The variable $HOLDS\_LOCK_i$ indicates which task holds the lock at file F_i. Procedures are common to all request handlers are: SEND (send the message *msg* to tasks T_j); INSERT (insert task T_j into a priority ordered queue *Q*); REMOVE (remove task T_j from queue *Q*).

Access Request Handler. Receiving an *access* message from an instance, causes the file manager to execute the following algorithm. If the execution queue is empty the file manager inserts the instance into the candidate queue and informs it of its *ready* status. Otherwise, it inserts the instance into the waiting queue.

```
task_request_access(task Tj, file manager Mi)
   {
      if (Ei == NULL) {/* is the Execution queue empty */
         INSERT(Tj,Ci);  /* place task Tj into Candidate queue */
         SEND("ready",Tj); }
      else {
         INSERT(Tj,Wi); } /* place task Tj into Waiting queue */
   }
```

Algorithm 4-1: Delayed Insertion Access Request Handler

Schedule Handler. Upon receipt of a schedule message from task T_j, file manager M_i executes the

following algorithm. The file manager first inserts the task instance into the execution queue (in the order defined in section 4.1.1). If the task instance is then at the front of the execution queue, the file manager then checks if another task instance already hold the lock. If no other task instance holds the lock, the file manager will notify the task instance that it has the lock. Otherwise, the file manager will request to call-back the lock from the lower priority task that currently holds the lock.

```
task_request_schedule(task Tj, file manager Mi)
   {
      INSERT(Tj,Ei);/* Schedule task Tj into the Ei for file Mi */
      if (front(Ei) == Tj) { /* is task Tj at the front of the Ei */
         if  (HOLDS_LOCKi == NULL) { /* is the lock free */
            SEND("lock_grant",Tj);/* send lock grant to task Tj*/
            HOLDS_LOCKi = Tj; }
         else {
            SEND("call_back",HOLDS_LOCKi); } } /*Call Back lock*/
   }
```

Algorithm 4-2: Delayed Insertion Schedule Handler

Release Handler. When the file manager M_i receives a release message from task T_j, it executes the algorithm task_request_release. The algorithm first removes the task instance from which the queue that it is currently located. It then checks if the highest priority task instance queued at the file is located in the waiting queue. If true, the algorithm then removes the task instance from the waiting queue and inserts this task instance into the candidate queue and sends to the task instance a *ready* message. The algorithm then checks if the lock is free. If free, the algorithm checks if the execution queue is empty. An empty execution queue causes the file manager to send *ready* messages to all task instances in the waiting queue and insert them in the candidate queue. If the execution queue is not empty, the file manager grants the lock to the highest priority task instance.

```
task_request_release(task Tj, file manager Mi)
   {
      if (Tj is in Wi) { /* is task Tj queue in the Wi */
         REMOVE(Wi,Tj); }/* remove task Tj from the Wi */
      else {
         if (Tj is in Ci) { /* is task Tj queue in the Ci */
            REMOVE(Ci,Tj); } /* remove task Tj from the Ci */
         else {
            REMOVE(Ei,Tj); } } /* remove task Tj from the Ei */
      if (the highest priority current client Tc of Mi is in Wi) {
         REMOVE(Tc,Wi);/* remove the task client Tc from the Wi */
         INSERT(Tc,Ci);/* insert the task client Tc into the Ci */
         SEND("ready",Tc); }
      if (HOLDS_LOCKi == NULL) { /* is the lock free */
         if (Ei == NULL) { /* is the Execution Queue empty */
            SEND("ready",Wi);/* send ready to tasks in Wi */
            Ci = UNION(Ci,Wi);/* place all Wi into the Ci */
            Wi = NULL;/* empty the Wi */ }
         else {
            HOLDS_LOCKi = Tj;/* send lock grant to Ei */
            SEND("lock_grant",front(Ei)); } }
   }
```

Algorithm 4-3: Delayed Insertion Release Handler

Call Back Acknowledgment Handler. When a call back acknowledge message is received by the file manager M_i it executes the following algorithm. It the execution queue is empty, the file manager will grant the lock to the task instance in the front of the execution queue.

```
task_request_acknowledge_callback(task Tj, file manager Mi)
   {
      if (Ei == NULL) { /* is the execution queue empty */
         HOLDS_LOCKi = Tj;
         /* send lock grant to the front task in the Ei */
         SEND("lock_grant",front(Ei)); }
   }
```

Algorithm 4-4: Delayed Insertion Call Back Acknowledgment Handler

4.1.2 File Server Client Allocation Policies

The FS client completes requests for the allocation of files needed by local task instances depending on the type of the access requested. Requests are separated into two types of categories: read access requests and write access requests. Read access requests are requests that execute preferably against a shadow copy, but could execute against a local private copy. The decision whether to use a private or shadow copy depends on the task instance's relative degree of sensitivity and relative degree of criticality. Write access requests are requests from task instances that intend to execute an update on all public copies.

Read Allocation Handling. The FS server requires that FS clients queue read requests at the site (where a copy resides) to ensure that the physical copy is not deleted before the read can be completed. FS clients, as a result, then handle read access requests (protocol graphically shown in figure 4-2) using the following steps:

1: Determine the location of either a shadow or local private copy based on the following priority:
   - Shadow copy, if a copy is locally available;
   - Private copy, if a copy is locally available and either of the following conditions are true:
     - The task instance is considered as robust;
     - The task instance is considered essentially critical;
   - Shadow copy, located at a remote site;

2: Requests to acquire the file copy from the FS server at the specified location (see step 1);

3: Complete the read request by sending a RPC to the server at the specified location.

4: Mark task instance successful/failed based on deadline of the task instance.

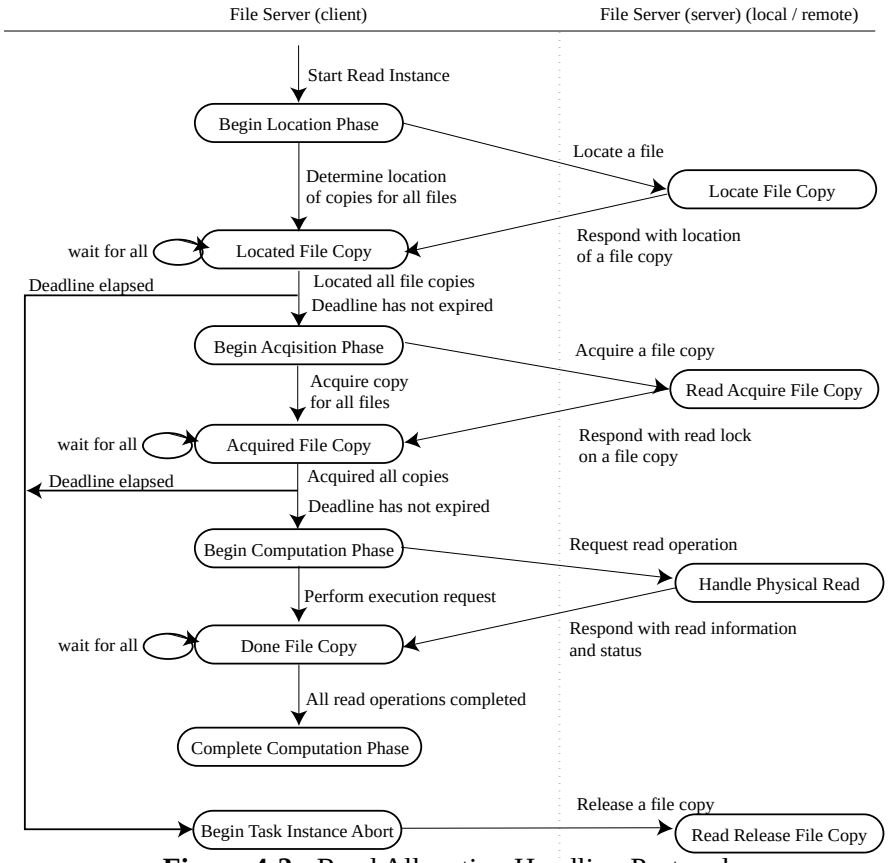


Figure 4-2: Read Allocation Handling Protocol

In step 1, the FS locates a readable (shadow/private) copy for all the files in task instance T_{jk}'s LRF_{jk} by selecting a copy based on T_{jk}'s location and T_{jk}'s C_{jk} and R_{jk}. Upon completion of step 1, the location of all files in T_{jk}'s LRF_{jk} has been fixed and a reasonable estimated execution time (EET_{jk}) for the read access is known. The FS client then goes on to step 2. In step 2, the FS client sends requests to acquire the file copies, determined in step 1, for T_{jk}. The FS client then waits to receive *acquired* messages from

the FS servers at the sites where the file copies reside. Once the FS client has received *acquired* messages from all files in T_{jk}'s LRF_{jk}, it then goes on to step 3. In step 3, the FS client requests that a read (specified by the application) be performed against the file copies specified in T_{jk}'s LRF_{jk}. The FS client then waits to receive *done* messages from the FS servers at the sites where the file copies reside. Upon receiving a *done* message from all needed file copies the File Server marks the read access request as completed. The FS client then checks if T_{jk} was successful in completing before its deadline (based on the definition of a successful read task in section 1.3).

Write Allocation Policy. Write access requests are handled (protocol graphically shown in figure 4-3)by the FS client using the following steps:

1: Determine the exact location of all public copies.
2: Request to be placed into the candidate queue at all file copy locations;
3: Request to be scheduled into the execution queue at all file copy locations;
4: Complete the write request by sending a RPC to all servers at the specified locations.
5: Mark task instance successful/failed based on deadline.

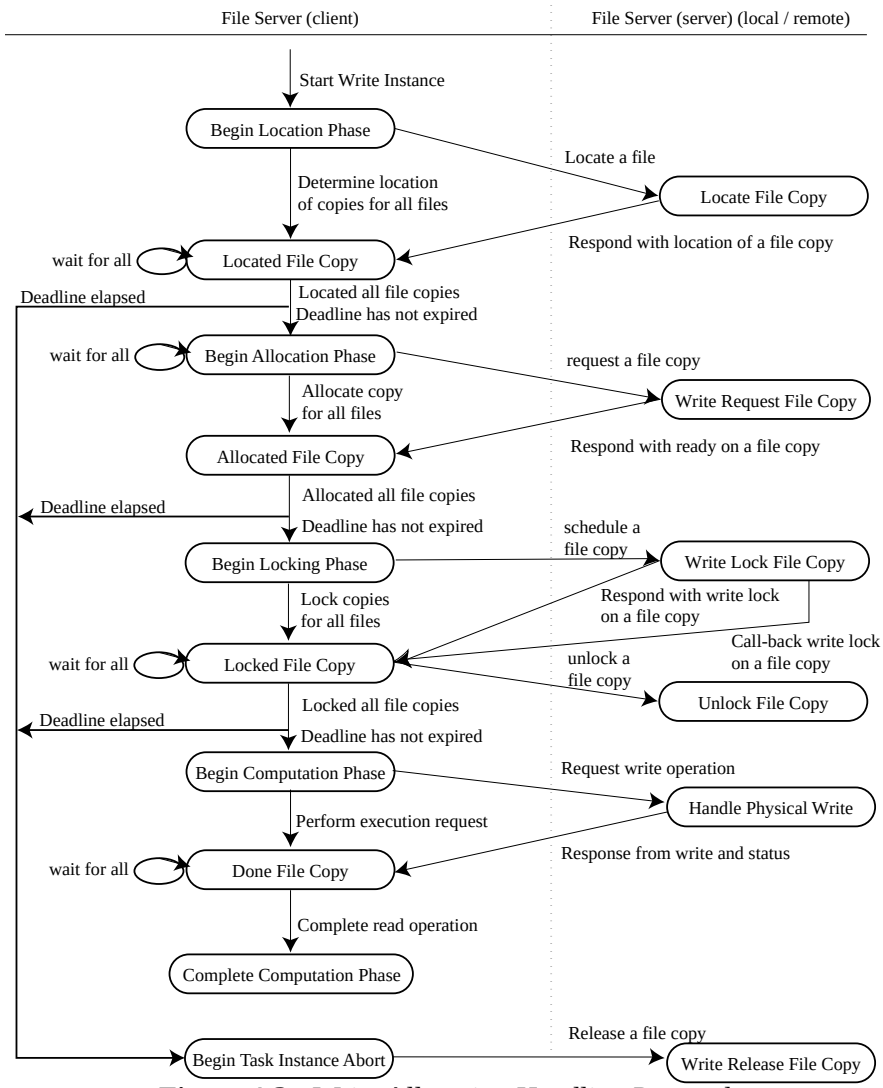


Figure 4-3: Write Allocation Handling Protocol

In Step 1, the FS client determines the location of all public copies of needed files in task instance T_{jk}'s LRF_{jk}. Upon the completion of Step 1, the location of all copies is fixed and a reasonable estimated execution time (EET_{jk}) for the write access is known. The FS client then goes on to step 2. In step 2, the FS client requests from all locations that T_{jk} is to be placed into the candidate queue for the public copy located at that site. The FS client receives a *ready* message from the FS server at a location once the task instance has been placed into the candidate queue at that location. Once all *ready* messages have

been received, the FS client then goes on to step 3. In step 3, the FS client requests from all locations that T_{jk} is to be scheduled into the execution queue for the public copy at the corresponding site. The FS client then receives a *lock-grant* message from the location once the T_{jk} is located at the front of the execution queue for the public copy. If during step 3, a *callback* request message is received, then the FS client marks the request as having released the lock and responds with a *callback-acknowledgment* message to the corresponding FS server. The FS client will only proceed to step 4 when all *lock-grant* messages have been received. In step 4, the FS client request that a write be performed on both the public and shadow copies at all locations. The FS client receives from a remote FS server a *done* message once the public and shadow copies have been successfully updated at the corresponding location. Included in the *done* message is the status, a timestamp, of when the write was completed. Once all *done* messages have been received, the FS client marks the write request as completed. The FS client then checks if the T_{jk} completed successfully or not (based on the definition of a successful write task given in section 1.3).

4.1.3 File Server Server Allocation Policies

The FS server receives requests from local and remote FS clients for the allocation of copies of files (located at the site) to a task instance T_{jk} (at the site of the FS client). Requests are separated into two categories critical access requests (CAR) and normal access requests (NAR). Critical access requests are those requests from T_{jk}s that are considered essentially critical. Critical access requests must be handled as soon as possible due to the urgency of T_{jk} and its direct influence on the survivability of the entire system. Normal access requests, on the other hand, are requests from task instances considered either critical or non-critical. The method used to determine which type of request is to be handled by the FS server is described in the section File Server/File Assigner integration. The policy for how to handle CAR or NAR types is only dependent on whether the request is for read/write access to the file copy.

Read Allocation Handling. The FS server allows for concurrent read access to a public copy by directing all read access requests to the corresponding shadow copy. A private copy located at the site provides faster read access without the additional overhead occurred by shadow copies. However, the information contained in the private copy is not guaranteed to be accurate. This inaccuracy results in only a limited number of task instances being allowed to access the private copy. Private copies only handle requests for read access given directly by the local FS client. A shadow copy is in effect a shared object in the MELODY system. Therefore, the FS server provides access for both the local and remote FS clients. The FS server requires that FS clients queue their read access. This ensures that the corresponding physical file is not deleted from the site while there are still outstanding read access requests for the copy located at the site. The FS servers ensure that the updating of a shadow copy is an atomic action that directly follows the update of the corresponding public copy. Therefore, FS servers are able to ensure that a request for read access to a shadow copy is at most one update behind the public copy. However, no such guarantee can be made on the information read from a local private copy. Since, there is no requirement to update the private copy immediately (only eventually) following the update of a public copy. For read access, to a locally available shadow or private copy, the FS server responds to the following requests by performing the corresponding action:

Acquire: The FS server marks the file copy as having been acquired for read access by queueing the requesting T_{jk} into its access queue. Once T_{jk} is queued, the FS server sends an *acquired* response message to the requesting FS client informing it that T_{jk} has acquired the file at its location. A request to acquire a private copy from a remote FS client is immediately denied by the FS server.

Execute: Upon receipt of a read *execute* request the FS server begins the physical read operation to access the physical file. Once completed, the FS server responds to the requesting FS client with the result of the read operation. The response is accomplished by sending a *done* message to the FS client requesting the read. Finally, the FS server removes T_{jk} from the file copies access queue.

Release: When the FS server receives a *release* request, it removes T_{jk} from the access queue.

Write Allocation Handling. Handling of write access requests to a public file copy, by the FS server, uses the Delayed Insertion protocol. The FS server responds to the following requests by an FS client (for a task instance T_{jk}) for write access to a public file copy by performing the corresponding action:

Request: The FS server places T_{jk}'s write request into either the candidate or waiting queue of the *Delayed Insertion* protocol. If T_{jk} was placed into the candidate queue, then the FS server immediately sends *ready* response. However, if T_{jk} is placed into the waiting queue, then the FS server sends a *ready* response to the requesting FS client when execution queue is empty and T_{jk} has been moved from the waiting queue into the candidate queue.

Schedule: The FS server moves T_{jk}'s the write request from the candidate queue into the execution queue of the *Delayed Insertion* protocol. A *lock-grant* response is only sent once T_{jk}'s write request is located at the front of the execution queue. A call-back message is sent to the current *lock-grant* holder according to the rules regarding callbacks in the *Delayed Insertion* protocol.

Call-Back Acknowledge: The FS server gives the lock to the front of the execution queue, and sends a *lock-grant* message to the player T_{jk} located in the front of the execution queue.

Execute: Upon receipt of a write execute request, the FS server begins the physical write operation to update the public and shadow copies located at the site. Only after the public copy has been updated, will the FS server respond with the result of the write operation using the *done* message. Included in the response is a timestamp of when the request was completed. This information is used by the FS client to determine if T_{jk} was able to successfully complete all requests before its deadline (DT_{jk}). Finally, it removes T_{jk} from the execution queue. After which, the FS server gives the lock to the next task instance located in the front of the execution queue, or move task instances from the candidate queue into the execution queue.

Release: When the FS server receives a *release* request, it removes T_{jk} from the waiting, candidate or execution queues.

4.1.4 Physical File Access Handling

To be able to guarantee, within established bounds, the physical execution time to a local copy, the FS server maps all files directly into memory and pins the associated memory. This guarantees that the data accessed during the read/write request is always in memory, and has a relatively fixed access time. The FS server receives all messages for the updating and reading of any file copy located on the site from either local/remote FS clients. An update operation is initially applied to the public copy located at the site. Once completed on the public copy, by use of an atomic action the FS server applies the update to the shadow copy (see example in section 3.2). Only after the shadow copy has completed the update, will the FS server respond to the requesting FS client that the update has completed. Included in this response is the time at which the update operation was completed. The success of failure of a task instance is then based upon this time using the rules associated to whether the write task instance was successful (see section 3.1). A read operation is executed on the corresponding shadow or private copy. The FS server responds to the FS client with the results of that execution and the time at which it was completed. However, the time is not utilized by the remote FS client. Since, the status of a read can only be determined by the requesting FS client, and the time at which the read done message was received by that location (see section 3.1).

4.1.5 File History

The FS server is designed to maintain the following file history information for every file accessed by local tasks or file copies located at the local site: **Sensitive Read Queue (FSR)** contains reading task instances that have failed and were considered either sensitive or essentially sensitive; **Robust Read Queue (FRR)** contains reading task instances that were considered robust; **Write Local Queue (FSW)** contains writing task instances that were unable to successfully complete prior to their deadline. The file history also contains information regarding the total number of local read access requests and the total number of write access requests (both local and remote) for each file. This information is

maintained for a predetermined amount of time. Thus it represents a set of task instances that have succeeded or failed during a fixed window of time.

This file history information could be utilized by the File Server to determine how to invoke the File Assigner to change the distribution of public copies in order to improve the performance of local tasks. For example, if the number of task instances queued in the *sensitive read queue* (*FSR*) surpassed a predetermined threshold value the File Server could then request that the File Assigner create a local public copy of this file in order to provide faster access to this file. For the tasks queued in the *write local queue* (*FSW*) the File Server could request that the File Assigner reduce the number of public copies in the system in order to reduce the time required to access the remaining copies. In case of tasks queued in the *robust read queue* (*FRR*) the File Server requests that a private copy would be created since this would improve their performance while not significantly impacting the performance of writing tasks within the system. Details regarding this type of File Assigner integration (**File Server-Oriented Integration**) will be described in section 6.1.4 as one of two distinctly different integration methods.

4.1.6 Task History

The FS client maintains the following task history information on the last five task instances for every task defined at the local site: **Status** (whether the instance was successful or failed); **Criticality** (relative degree of criticality of the instance; **Sensitivity** (relative degree of sensitivity of the instance; **Location Time** (time required to locate all required files copies by the instance); **Acquisition Time** (time required by the instance to obtain all required locks); **Computation Time** (time to completed the requested read/write request by the instance). This history information allows the Run-Time Monitor (chapter 7) to determine the relative degree of criticality and sensitivity of newly arrived task instances. Also, the Task Scheduler (see section 5.1) uses the information to determine a reasonable estimate for the estimated execution time of a scheduling task instance.

4.2 File Server Implementation

As a summary of the previous presentation the File Server the functionality of the File Server is given in pseudo-code in this section.

4.2.1 File Server Client Implementation

Services provided by the FS client can be categorized into the two main responsibilities: task handling and file response handling. Task handling services included all services to control the progression of a task instance from one phase in the task life cycle (see section 3.1) to another phase (detailed algorithms for the following routines can be found in section 4.2.1.1). File response services include those services that directly handle the responses from either the local/remote FS servers regarding a copy located at the site. This response includes or designates the status of task instance's request for access to that specific file copy (detailed algorithms for the following routines can be found in section 4.2.1.2):

4.2.1.1 File Server Client Task Handling Services

Each of the FS client services handles one specific phase of a task instance T_{jk}'s task life cycle. Each of these services take as a parameter the task instance to be handled by the routine.

Begin Location Phase. This service locates all required copies of files in task instance T_{jk}'s list of required files (LRF_{jk}). For a reading T_{jk} this service locates either a locally available shadow or private copy or a remote shadow copy. For a writing T_{jk} this service locates all public copies. In algorithm 4-5, the instance T_{jk} is marked as having started its location phase. The algorithm uses the conditional expression IS_WRITE_ACCESS_REQUEST to determine whether the type of access requested is write access. If true, then the algorithm makes a request to locate all public copies. Otherwise, the conditional expression IS_LOCAL_COPY_AVAILABLE determines if there is a copy (based on definition given in

section 1.3) at the site that could be used. If there is a locally copy, then the algorithm makes a request to complete the location of the local private or shadow copy of file F. Otherwise, the algorithm requests to locate a shadow copy of file F at a remote site.

```
Begin Location Phase(Task_Instance Tjk)
{
   mark_task_instance(Tjk,PHASE_LOCATION);
   for all request R in Tjk's LRFj do {
      if IS_WRITE_ACCESS_REQUEST(R) {
         request_locate_public_copies(R,Tjk); }
      else {
         if IS_LOCAL_COPY_AVAILABLE(R,Tjk) {
            request_locate_local_copy(R,Tjk); }
         else {
            request_locate_remote_shadow_copy(R,Tjk); } } }
}
```

Algorithm 4-5: Begin Location Phase

Begin Acquisition Phase. Whether the task instance T_{jk} is a writing or reading task instance is first determined by this service. For a writing task instance it begins the allocation sub-phase. For a reading task it proceeds to go ahead and make the requests to acquire the file copies located during the T_{jk}'s location phase. In algorithm 4-6, the task instance T_{jk} is marked that it has started its acquisition phase. The conditional expression IS_TASK_WRITING_TASK determines whether T_{jk} is writing to the files in its LRF_{jk}. If true, the algorithm begins the allocation sub-phase (applicable only to writing instances). Otherwise, the algorithm requests to acquire all copies for T_{jk}. This request to acquire either a shadow or private copy is sent as a single (point-to-point) message to the FS server at the specific site located during T_{jk}'s location phase.

```
Begin Acquisition Phase(Task Instance Tjk)
{
   mark_task_instance(Tjk,PHASE_ACQUISITION);
   if IS_TASK_WRITING_TASK(Tjk) {
      Begin Allocation Phase(Tjk); }
   else {
      for all request R in Tjk's LRFj do {
         request_to_aquire_file_copy(R); } }
}
```

Algorithm 4-6: Begin Acquisition Phase

Begin Allocation Phase. For a writing task instance T_{jk}, this service makes the requests to allocate those files to T_{jk}. Meaning that the T_{jk} requests to be placed into the candidate queue at all locations holding a public copy of the file. The FS client uses algorithm 4-7 to mark that task instance T_{jk} has started its Allocation sub-phase. The algorithm requests for all file copies that T_{jk} be placed into the candidate queue at every location that holds a public copy. This request is sent as a multicast request to all FS servers that have a public copy of the requested file. The location of these sites would have been determined during T_{jk}'s location phase.

```
Begin Allocation Phase(Task Instance Tjk)
{
   mark_task_instance(Tjk,PHASE_ALLOCATION);
   for all request R in Tjk's LRFj do {
      request_to_allocation_file(R); }
}
```

Algorithm 4-7: Begin Allocation Phase

Begin Locking Phase. For a task instance T_{jk}, that has completed its allocation sub-phase (allocated all files in its LRF_{jk}), this service makes the requests to obtain a write lock on those public copies of files for T_{jk}. The FS client uses algorithm 4-8 to mark that task instance T_{jk} has started its locking sub-phase. The algorithm requests for all file copies in T_{jk}'s LRF_{jk} that T_{jk} be placed into the execution queue at every location that holds a public copy. This request is sent as a multicast request to all FS servers that have a public copy of the requested file located at that site. The location of these sites would have been allocated to T_{jk} during its allocation phase.

```
Begin Locking Phase(Task Instance Tjk)
{
   mark_task_instance(Tjk,PHASE_LOCKING);
   for all request R in Tjk's LRFj do {
      request_to_lock_file(R); }
}
```

Algorithm 4-8: Begin Locking Phase

Begin Computation Phase. A task instance T_{jk}, that has completed its acquisition phase (writing task instances would have completed both Allocation and locking sub-phases) and has either acquired (reading task instances) or locked (writing task instances) all files in its LRF_{jk}, use this routine to requests to perform either the read/write request on their files. In algorithm 4-9, the task instance T_{jk} is marked that it has started its computation phase. The conditional expression IS_TASK_WRITING_TASK determines whether T_{jk} is writing to the files in its LRF_{jk}. If true, then the algorithm requests for all file copies in the LRF_{jk} that they update the public and shadow copy locked for this task instance. The request is sent so that all FS servers detect that T_{jk} is updating the public copies of file F. This is required so that those FS servers with private copies may also detect the update and proceed accordingly in determine when to refresh their local private copy. If T_{jk} is a reading, then the FS client makes requests, for all file copies in T_{jk}'s LRF_{jk}, that they complete the requested read request for T_{jk}. This request is sent as a single message since no other FS server needs to be informed of the read request. The copies requested to perform the read request would have already marked the file as acquired by T_{jk}. This marking ensure that the file will not be deleted until at least this T_{jk} has completed its read request.

```
Begin Computation Phase(Task Instance Tjk)
{
   mark_task_instance(Tjk,PHASE_COMPUTATION);
   if IS_TASK_WRITING_TASK(Tjk) {
      for all request R in Tjk's LRFj do {
         request_to_update_file(R); }
   else {
      for all request R in Tjk's LRFj do {
         request_to_read_file_copy(R); }
}
```

Algorithm 4-9: Begin Computation Phase

Complete Computation Phase. Once the task instance T_{jk} has completed its read/write request, the FS client determines if it was successful and updates the task history information with that information. After updating the task history, the FS client finishes T_{jk}'s task life cycle by removing it from the system. In algorithm 4-10, the T_{jk} is marked that it has completed its computation phase. The conditional expression IS_TASK_WRITING_TASK determines whether the T_{jk} is writing to the files in its LRF_{jk}. If true, then the FS client checks whether T_{jk} successfully updated the files according to the definition of a successful writing task (see section 1.3). If successful, then the FS client updates the task history information that T_{jk} has completed successfully. Otherwise, the algorithm the task history information is updated with a failed status. For reading instances, the FS client determines whether T_{jk} successfully completed the read request according to the definition of a successful reading task (section 3.1). If successful, then the task history information is updated that T_{jk} has completed successfully, otherwise it's updated that T_{jk} failed. Once the task history has been updated, the FS client proceeds to finish any required processing to remove T_{jk} from the MELODY system.

```
Complete Computation Phase(Task Instance Tjk)
{
   mark_task_instance(Tjk,PHASE_COMPLETED);
   if IS_TASK_WRITING_TASK(Tjk) {
      if WAS_WRITE_SUCCESSFUL(Tjk) {
         update_task_history(SUCCESSFUL,Tjk); }
      else {
         update_task_history(FAILED,Tjk); }
   else {
      if WAS_READ_SUCCESSFUL(Tjk) {
         update_task_history(SUCCESSFUL,Tjk); }
      else {
         update_task_history(FAILED,Tjk); } }
   Finish Task Instance(Tjk);
}
```

Algorithm 4-10: Complete Computation Phase

Begin Task Instance Abort. For a task instance T_{jk}, that has been aborted during either its location or acquisition phases (task instances can not be aborted during any other phase) the FS client determines whether there are any requests that need to be released by the T_{jk}. This determination is based on whether T_{jk} was trying to acquire shadow or private copies, or it was trying to either allocate or lock public copies. If no requests need to be released, the FS client finishes the task life cycle for T_{jk}. Otherwise, the task instance requests to release those outstanding requests before removing T_{jk} from memory and the MELODY system. In algorithm 4-11, the task instance T_{jk} is marked that it has been aborted by the MELODY system. The conditional expression HAS_INSTANCE_COMPLETED_LOCATION_PHASE determines if T_{jk} had completed the location of the files in its LRF_{jk}. If T_{jk} had completed its Location Phase, then the FS client knows that there are outstanding requests that need to be released before the FS client can finish T_{jk}. The conditional expression IS_TASK_WRITING_TASK determines whether T_{jk} was attempting to write to the copies of files in its LRF_{jk}. If T_{jk} was a writing task instance, then the algorithm makes requests for all file copies in T_{jk}'s LRF_{jk} that they release T_{jk} from any of the delayed insertion queue (waiting, candidate or execution) in which it may be queued currently. This request is sent as a multicast request to those files located during T_{jk}'s location phase. If T_{jk} is a reading task instance, the algorithm makes requests to release all copies of files in T_{jk}'s LRF_{jk} by sending single (point-to-point) messages to those specific FS server locations determined during the T_{jk}'s location phase.

```
Begin Task Instance Abort(Task Instance Tjk)
{
   mark_task_instance(Tjk,PHASE_ABORTED);
   if HAS_INSTANCE_COMPLETED_LOCATION_PHASE(Tjk) {
      if IS_TASK_WRITING_TASK(Tjk) {
         for all request R in Tjk's LRFj do {
            request_to_release_file(R); } }
      else {
         for all request R in Tjk's LRFj do {
            request_to_release_file_copy(R); } } }
   Finish Task Instance(Tjk);
}
```

Algorithm 4-11: Begin Task Instance Abort

4.2.1.2 File Server Client File Response Services

Each of the FS client file response services handles a specific response from a FS server (local/remote) to a specific task instance T_{jk} regarding the status of its request for acquisition of a specific file copy F located at that site. These services all take the following parameters: T_{jk} (task instance wait for the response); F (file the response is concerning [must be a member of LRF_{jk}]); I (site of the FS server responding to a request).

Located File Copy. This service handles a response from an FS server that has a copy of the requested file. This copy can be used by the local T_{jk} based on the definition of files accessibility given in section 3.1. For reading T_{jk} this means that the FS server has a usable shadow or private (if the local FS server is responding) copy located at its site. A writing T_{jk} only receives responses from those FS

servers that have a public copy located at its site. The algorithm 4-12 finds the specific file request that the response is about in T_{jk}'s LRF_{jk}. Once the request has been located, the algorithm adds the location of this copy to the request. The algorithm then marks this copy as having been located. If the algorithm detects (using the conditional expression HAS_TASK_LOCATED_ALL_FILES) that all required copies of the files in T_{jk}'s LRF_{jk} have been located, then the FS client begins the task instance's acquisition phase by calling the FS client routine begin acquisition phase.

```
Located File Copy(Task_Instance  Tjk, File  F, Host  I)
{
   The_Request = find_request(F,LRFj);
   add_copy(I,The_Request);
   mark_file_copy(The_Request,I,LOCATED);
   if HAS_TASK_LOCATED_ALL_FILES(LRFj) {
      Begin Acquisition Phase(Tjk); }
}
```

Algorithm 4-12: Located File Copy

Acquired File Copy. A response from an FS server that it has queued task instance T_{jk} into its access queue is handled by this service. This acknowledgment guarantees that the FS server does not physically delete the copy at least until after T_{jk} has completed its computation phase or has released this copy after being aborted. This service is only invoked for reading task instance. Writing task instances never receive *acquired* message responses. The algorithm 4-13 first searches in T_{jk}'s LRF_{jk} for the specific file copy the response is concerns. The algorithm then marks the requested file copy as having been acquired. If the algorithm detects that all required copies of the files in T_{jk}'s LRF_{jk} have been acquired, then the algorithm begins T_{jk}'s computation phase by calling the FS client service begin computation phase.

```
Acquired File Copy(Task_Instance  Tjk, File  F, Host  I)
{
   The_Requested_Copy = find_request_copy(F,I,LRFj);
   mark_file_copy(The_Requested_Copy,ACQUIRED);
   if HAS_TASK_AQUIRED_ALL_FILES(LRFj) {
      Begin Computation Phase(Tjk); }
}
```

Algorithm 4-13: Acquired File Copy

Allocated File Copy. The response from an FS server at that it has queued task instance T_{jk} into the candidate queue (Delayed Insertion protocol) is handled by this service. The service is only invoked for a writing task instance. A reading task instance never receives *ready* message responses. The algorithm 4-14 first searches in T_{jk}'s LRF_{jk} for the specific file copy. The algorithm then marks the requested file copy as having been allocated. If the algorithm detects that all required copies of the files in T_{jk}'s LRF_{jk} have been allocated then the algorithm begins T_{jk}'s Locking sub-phase by calling the FS client service begin locking phase.

```
Allocated File Copy(Task_Instance  Tjk, File  F, Host  I)
{
   The_Requested_Copy = find_request_copy(F,I,LRFj);
   mark_copy_aquired(The_Requested_Copy);
   if HAS_TASK_ALLOCATED_ALL_FILES(LRFj) {
      Begin Locking Phase(Tjk); }
}
```

Algorithm 4-14: Allocated File Copy

Locked File Copy. This service handles a response from an FS server that has queued task instance T_{jk} into the execution queue (Delayed Insertion protocol) at the site. Receiving this response means that T_{jk} is currently located at the front of the execution queue and holds the lock on this specific public copy. This service is only invoked for a writing task instance. A reading task instance never receives *lock-grant* message responses. The algorithm 4-15 first searches in T_{jk}'s LRF_{jk} for the specific file copy the response is in regards to. The algorithm then marks the requested file copy as having been granted a write lock. If the algorithm detects that all required copies of the files in T_{jk}'s LRF_{jk} have been locked, then the algorithm begins T_{jk}'s computation phase by calling the FS client service begin computation phase.

```
Locked File Copy(Task_Instance  Tjk, File  F, Host  I)
{
   The_Requested_Copy = find_request_copy(F,I,LRFj);
   mark_copy_locked(The_Requested_Copy);
   if HAS_TASK_LOCKED_ALL_FILES(LRFj) {
      Begin Computation Phase(Tjk); }
}
```

Algorithm 4-15: Locked File Copy

Called-Back File Copy. This service handles a response from an FS server that it previously granted a write lock (*lock-granted* message), and now needs to callback that granted write lock. The reason for this is that a more critical task instance has been placed into Execution Queue (Delayed Insertion protocol) and is currently in front of the local task instance T_{jk}. In algorithm 4-16, the conditional expression IS_TASK_IN_LOCKING_PHASE determines if the T_{jk} is currently in the locking sub-phase. If T_{jk} is still trying to lock other copies, the algorithm allows the FS server to callback its previously granted write lock. The algorithm does this by marking the file copy as being unlocked (effectively setting the status back to as if it had been only allocated, therefore never having received a *grant-lock* message from this FS server). Then the algorithm sends a single (point-to-point) request to the FS server at site I acknowledging that it has given up the write lock on the public copy of file F located at that site. If the task had not been in the Locking sub-phase, the algorithm ignores the response to callback the granted write lock. This denial to give up the write-lock assumes that the task is already in the computation phase, and that it would be better (no rollbacks or additional overhead) to just allow T_{jk} to complete its execution (therefore releasing the files normally) rather than trying to cancel the execution early.

```
Called-Back File Copy(Task_Instance  Tjk, File  F, Host  I)
{
   if IS_TASK_IN_LOCKING_PHASE(Tjk) {
      The_Requested_Copy = find_request_copy(F,I,LRFj);
      mark_copy_unlocked(The_Requested_Copy);
      acknowledge_callback(The_Requested_Copy); }
}
```

Algorithm 4-16: Called-Back File Copy

Done File Copy. The response from an FS server that it has completed the requested read/write request is handled by this service. Informing the task instance that the request from the task instance has been removed from its site. The algorithm 4-17 first finds the specific requested file copy in T_{jk}'s LRF_{jk} for file the F at site I. The algorithm then marks the requested file copy as having received a done message from the FS server at that site. If the algorithm detects that all required copies in T_{jk}'s LRF_{jk} have completed their request, then the algorithm begins to complete the computation phase for T_{jk} by calling the service complete computation phase.

```
Done File Copy(Task_Instance  Tjk, File  F, Host  I)
{
   The_Requested_Copy = find_request_copy(F,I,LRFj);
   mark_copy_done(The_Requested_Copy);
   if HAS_TASK_COMPLETED_AT_ALL_FILES(LRFj) {
      Complete Computation Phase(Tjk); }
}
```

Algorithm 4-17: Done File Copy

File Distribution Change. A responses from an FS server that it has changed the distribution of file F is controlled by this service. In algorithm 4-18, the first checks if T_{jk} is writing to the files in its LRF_{jk}. If T_{jk} is writing, the algorithm checks whether the change is to create a public copy. If a public copy has been created, then the algorithm adds the copy to T_{jk}'s LRF_{jk}. The algorithm then requests to refresh the information at that site with the information regarding T_{jk}. If the change is the deletion of a public copy, then the algorithm deletes the copy from T_{jk}'s LRF_{jk}. All other changes in the distribution of copies of file F do not effect the completion of writing task instance T_{jk}. The algorithm handles a reading task instance T_{jk} by checking if the change was the activation of a local public file copy. If true, then the FS client deletes the old request to access a remote shadow copy or local private copy, and request to access the locally available shadow copy. This improves the chances for T_{jk} to successfully

complete before its deadline, while providing it with the most current file information.

```
File Distribution Change(Task_Instance  Tjk, File  F, Host  I, File_Change_Type  Type)
{
   if IS_TASK_WRITING_TASK(Tjk) {
      if (Type == CREATE_PUBLIC_COPY) {
         The_Request = find_request(F,LRFj);
         add_copy(I,The_Request);
         request_refresh_copy(I,The_Request); }
      elseif (Type == DELETE_PUBLIC_COPY) {
         The_Request = find_request(F,LRFj);
         delete_copy(I,The_Request); } }
   else {
      if ((Type == ACTIVATE_PUBLIC_COPY) AND IS_HOST_LOCAL(I)) {
         The_Request = find_request(F,LRFj);
         delete_copy(The_Request);
         add_copy(I,The_Request);
         request_refresh_copy(I,The_Request); } }
}
```

Algorithm 4-18: File Distribution Change

4.2.1.3 File Server Client Implementation of Task History

The FS client is required to maintain information on the last five task instances of task T_j regarding their: status, relative degree of criticality (C_{jk}), relative degree of sensitivity (R_{jk}), and execution time of T_{jk}. This task history information is then used by both the Run-Time Monitor (chapter 7) and Task Scheduler (chapter 5) for determining the C_{jk}, R_{jk} and EET_{jk} of future task instances. To accomplish this the FS client provides the history management services (described in the following paragraphs) to update or query the task history for a given T_j:

Update Task History. This service handles a request to update the task history of a task T_j with the success or failure of a task instance T_{jk}. The FS client maintains a predetermined amount of history information for every task T_j. In the current implementation a maximum of five stored history values is maintained. The algorithm 4-19 first determines if the maximum number of history items has been exceeded or not. If it has been exceeded, then the oldest history item is removed from the task history. The algorithm then adds the information regarding the new history item about task instance T_{jk} to the task history.

```
Update Task History(Task_Instance  Tjk, Task_History  Historyj)
{
   if (number_history_items(Historyj) >= MAX_TASK_HISTORY) {
      delete_history_item(Historyj5); }
   add_new_history_item(Tjk,Historyj);
}
```

Algorithm 4-19: Update Task History

Relative Criticality. The determination of the relative criticality C_{jk} of task instance T_{jk} based on the task history (containing the status and $C_{j(k-1)}$ of the previous task instance $T_{j(k-1)}$ for task T_j (see section 3.1)) is done by this service. The thresholds a_i' and a_i" are considered to be constant values and defined during an initialization phase. The defined initial critical C_j of T_j would be determined by the applications predefined definition of T_j. The algorithm 4-20 first determines if the last instance $T_{j(k-1)}$ was successful. If successful, the algorithm then returns the value of C_{jk} based on C_j. Otherwise, the service checks if the $T_{j(k-1)}$'s $C_{j(k-1)}$ was considered essentially critical (less than or equal to a_i'). If so, then threshold a_i' is returned as the value for T_{jk}'s C_{jk}. If $T_{j(k-1)}$ was not considered essentially critical, the algorithm then checks if $T_{j(k-1)}$'s $C_{j(k-1)}$ was considered non-critical (greater than or equal to a_i"). If so, then threshold a_i" is returned as the value for T_{jk}'s C_{jk}. If none of the previous conditions were true, then the value of ($C_{j(k-1)}$-1) is returned as the value for T_{jk}'s C_{jk}.

```
Relative Degree of Criticality(Task  Tj, Task_History  Historyj)
{
   if (Historyj->Last_Instance.Status == SUCCESSFUL) {
      return(initial_criticality(Tj); }
   else {
      if (Historyj->Last_Instance.Criticality <= ai' ) {
         return(ai'); }
      else {
         if (Historyj->Last_Instance.Criticality >= ai'' ) {
            return(ai''); }
         else {
            return(Historyj->Last_Instance.Criticality - 1); } } }
}
```

Algorithm 4-20: Relative Criticality of a Task Instance

Relative Sensitivity. In order to determine the relative sensitivity R_{jk} of task instance T_{jk} based on the task history (containing the status and $R_{j(k-1)}$ of the previous task instance for task T_j (see section 3.1)), the thresholds b_i' and b_i" are considered to be constant values and defined during an initialization phase. The defined initial sensitivity value of a T_j would then be determined by the application's definition. The algorithm 4-21 first determines if the last instance was successful. If successful, then the algorithm returns the value of R_{jk} based on R_j. Otherwise, the service checks if the $T_{j(k-1)}$ was considered essentially sensitive (less than or equal to b_i'). If so, then threshold b_i' is returned as the value for T_{jk}'s R_{jk}. If $T_{j(k-1)}$ was not considered essentially sensitive, the algorithm then checks if $T_{j(k-1)}$ was considered robust (greater than or equal to b_i"). If so, then threshold b_i" is returned as the value for T_{jk}'s R_{jk}. If none of the previous conditions were true, then the value of ($R_{j(k-1)}$ + 1) is returned as the value for T_{jk}'s R_{jk}.

```
Relative Degree of Sensitivity(Task  Tj, Task_History  Historyj)
{
   if (Historyj->Last_Instance.Status == SUCCESSFUL) {
      return(initial_sensitivity(Tj); }
   else {
      if (Historyj->Last_Instance.Sensitivity <= bi' ) {
         return(bi'); }
      else {
         if (Historyj->Last_Instance.Sensitivity >= bi'' ) {
            return(bi''); }
         else {
            return(Historyj->Last_Instance.Sensitivity + 1); } } }
}
```

Algorithm 4-21: Relative Sensitivity of a Task Instance

Estimated Execution Time. This service calculates the estimated execution time (EET_{jk}) (see section 3.1) for a task instance T_{jk} based on the task history (containing the status and execution time of a previous determined *n* task instances T_{j1}...T_{jn}) for task T_j. This information is then used by the local Task Scheduler during the scheduling of T_{jk}. The algorithm 4-22 determines the average execution time of all history items in the T_j's task history by summing them together and then dividing by the number of items. Then the routine for every file in the T_{jk}'s LRF_{jk} adds the computation time (including remote access to locate and acquire the files).

```
Estimated Execution Time(Task  Tj, Task_History  Historyj)
{
   EET = 0;
   for all history item in Historyj do {
      EET = EET + Historyjitem->Execution_Time; }
   EET = EET / (number history items in Historyj);

   for all files R in LRFj do {
      EET = EET + Computation_Time(R); }
}
```

Algorithm 4-22: Estimated Execution Time

4.2.2 File Server Server Implementation

The FS server module can be viewed as four separate sub-servers: acquisition, execution, history

and movement; each controlling one specific aspect regarding the copies of files located at the site (figure 4-4). The responsibilities for each sub-server are defined as follows:

Acquisition: Is responsible for controlling the acquisition (including allocation and locking) for copies of files located at the site by FS clients (local and remote);

Execution: Handles the physical read/write operations requested by an FS client (local/remote) to a specific physical data file. The execution sub-server is also responsible for ensuring that the shadow copy is updated in conjunction with the update of the public copy;

History: Monitors the access for all files accessed by local tasks, or file copies located at the site. File Assigner (local/remote) access this history to determine how to change the file distribution;

Movement: Physically transfers a copy of a file to (from) the local site from (to) a remote site.

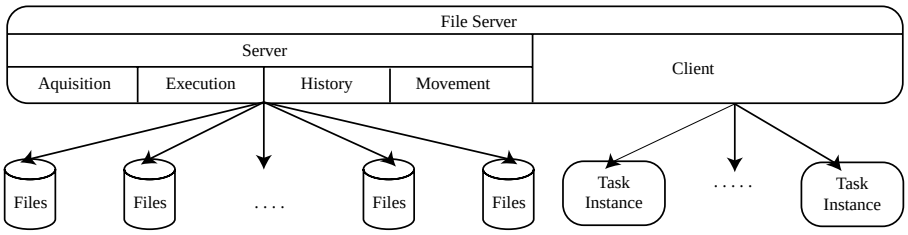


Figure 4-4: File Server Client / Server Modules

4.2.2.1 File Server Server Implementation of Files

MELODY provides three types of file copies (public, shadow and private) to be distributed amongst the sites in the MELODY system (see section 3.2). The local FS server implements these files as one file object that contains a file manager for each file type (figure 4-5). These managers are then responsible for handling the access request for that specific file type. The public file manager controls all write access to the public copy (also handling the atomic update of the shadow copy), according to the rules established by the *Delayed Insertion* protocol. The other two managers (shadow and private) are only concerned with read access and the guaranteeing that the physical file remains accessible while pending read requests (from local or remote FS clients) have been queued at this specific shadow or private copy. The status of a file manager, determined by the state of physical file information located at the site, is set to either: *active, partial, deleted* or *inactive*. An **active** status means that the associated physical file located at the site is available (contains all the physical data) to the FS clients (local/remote) for acquisition and access. A status of **partial** is used to state that the file associated with this manager has recently been created at the site. However, the transfer of the physical file data has not been completed by the FS Movement module. A public file manager with a partial status must respond to all allocation and locking request to ensure data consistency once the file transfer is completed (see section 4.2.2.5). A shadow or private file manager with a partial status does not respond to any access request, since it can not guarantee that data read from this site would be available to the requesting site when requested. The **deleted** status means that the FS server has received a request to delete the associated data file from a File Assigner (local/remote). However, there are pending requests for access that must be handled before the physical deletion of the file can be completed. The **inactive** status states that this file manager has no file located at the site (will not respond to any requests).

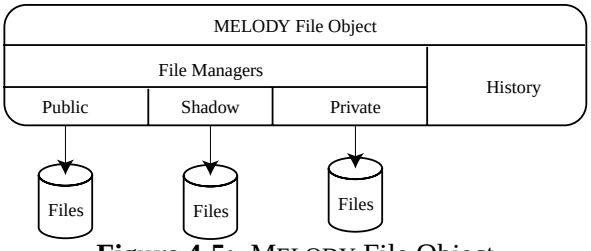


Figure 4-5: MELODY File Object

The file objects also contain a file history information regarding: **access type** (type of access read/write); **success rate** (number of successful requests to access this file); **failure rate** (number of failed requests to access this file). All file history information is maintained for a certain predetermined

interval of time. This interval of time is specified during the initialization phase of the MELODY system.

4.2.2.2 File Server Server Implementation of Acquisition Services

Acquisition services provided by the FS server are separated by the type of access (read or write) requested by an FS client (local or remote). Read acquisition services are handled by either the shadow or private file manager depending upon the location of the requesting FS client, and the status of the file managers. Write acquisition services are handled by the local public file manager. The implementation of these services is described in the following paragraphs.

Read Acquire File Copy. This service provided by either the shadow or private file manager places the requesting task instance T_{jk} into an access queue of requesting players. It responds to the requesting FS client that the file has granted a read request at the local site. In algorithm 4-23, the FS server tries to locate an active file manager for the file F requested by the task instance T_{jk} located at site I. If the FS server can not locate an active file manager, then it does nothing in responding to the request from the FS client at site I. If an active file manager was found, it then checks if the type of file manager found was a private file manager. If this is true and the site of the requesting T_{jk} is not local, then the FS server does nothing in responding to the request from the FS client at site I. Otherwise, the FS server queues the acquisition request and responds that T_{jk} has acquired the requested file at this site.

```
Acquire Read Copy(Task_Instance  Tjk, File  F, Host  I)
{
   A_Manager = locate_active_manager(F);
   if (A_Manager->Status == ACTIVE) {
      if ((A_Manager->Type == PRIVATE) AND (I is not local Host)) { }
      else {
         queue_access_request(Tjk,I);
         grant_acquisition_request(Tjk,I); } }
}
```

Algorithm 4-23: Read Acquire File Copy

Read Release File Copy. Provided by either the shadow or private file manager, this service removes the request made by task instance T_{jk} at site I from the file manager's access queue of requesting task instances that have been granted read access to the associated file copy. The algorithm 4-24 removes a previously queued request from T_{jk} at site I by first locating the file manager for the associated access type. The algorithm then dequeues T_{jk} from its access queue. No acknowledgment is sent, since all access request messages expire after their associated deadline. Therefore, there is no requirement to guarantee (by sending release acknowledgment messages) that the release action completed at a certain site.

```
Release Read Copy(Task_Instance  Tjk, Host  I, Access_Type  Type, File  F)
{
   A_Manager = locate_file_manager(F,Type);
   dequeue_access_request(Tjk,I);
}
```

Algorithm 4-24: Read Release File Copy

Write Request File Copy. The local public file manager queues the associated task instance T_{jk} into either the candidate or waiting queue based on the rules established by the Delayed Insertion protocol using this service. The algorithm 4-25 handles the write access request by completing the following steps. First it locates the public file manager at the site for the file F requested. If the manager is active or partial, it proceeds to determine whether the execution queue is empty or not. An empty execution queue allows the file manager to place the new request from T_{jk} directly into the candidate queue of the Delayed Insertion protocol (see section 4.1.1). If T_{jk} is placed into the candidate queue, the algorithm sends a response (*ready* message) that T_{jk} has acquired this file copy. Otherwise, the file manager queues T_{jk} into the waiting queue.

```
Request Write Copy(Task_Instance  Tjk, File  F, Host  I)
{
   A_Manager = locate_public_manager(F);
   if ((A_Manager->Status == ACTIVE) OR (A_Manager->Status == PARTIAL)) {
      if (A_Manager->Execution is Empty) {
         queue_player_request(Tjk,I,A_Manager->Candidate);
         send_player_ready(Tjk,I); }
      else {
         queue_player_request(Tjk,I,A_Manager->Waiting); } }
}
```

Algorithm 4-25: Write Request File Copy

Write Lock File Copy. A local public file manager (see figure 4-5) uses this service to move a previously queued task instance T_{jk} from the candidate queue into the execution queue. Afterwards, the file manager checks to see if T_{jk} is at the front of the execution queue. If T_{jk} is at the front, the file manager then checks if the lock has already been granted to another task instance. If so, the file manager checks whether the lock needs to be called back (sending a *callback* response message to the current lock holder) based on the criticality and deadline of the two instances. If there currently is no lock on the file, the file manager then grants to the requesting task instance the lock for this file copy. For T_{jk} that is not at the front of the execution queue, the file manager does no more additional work. The algorithm 4-26 handles requests from task instance T_{jk} to lock the public copy of file F located at the site. First it locates the public file manager at the site for the file requested. If the manager is active or partial, it proceeds to dequeue T_{jk} from the candidate queue and schedule it into the execution queue. It then checks to see if T_{jk} has been placed into the front of the execution queue (based on T_{jk}'s criticality and deadline). If so, it checks to see if the lock has already been granted to another task instance. If no lock has been previously been granted, then it marks the task instance as having the lock. The algorithm, after granting the lock to T_{jk}, sends to T_{jk} a response that it has been granted the lock (*lock-granted* message) at this file copy. If another task instance has already been granted the lock, the file manager requests to *call back* (see section 4.1.1) the file lock from that task instance. In all other cases, the file manager does nothing more.

```
Lock Write Copy(Task_Instance  Tjk, File  F, Host  I)
{
   A_Manager = locate_public_manager(F);
   if ((A_Manager->Status == ACTIVE) OR (A_Manager->Status == PARTIAL)) {
      dequeue_player_request(Tjk,I,A_Manager->Candidate);
      queue_player_request(Tjk,I,A_Manager->Execution);
      if (Tjk in front of A_Manager->Execution) {
         if (A_Manager->Holds_Lock == NULL) {
            A_Manager->Holds_Lock = Tjk;
            send_player_grant_lock(Tjk,I); }
         else {
            send_player_call_back(A_Manager->Holds_Lock); } } }
}
```

Algorithm 4-26: Write Lock File Copy

Write Unlock File Copy. This service handles the acknowledgment of a call back (see section 4.1.1) made by the task instance T_{jk} at site I that currently holds the lock on the public copy. After unlocking the file, it sends a grant lock to the task instance at the front of the execution queue. The algorithm 4-27 responds to a request by task instance T_{jk} that previously had been granted the write lock, and is acknowledging a callback request (resulting from a higher priority task instance being queued in front of T_{jk}). The FS server first locates the public file manager at the site for the file requested. If the manager is active or partial, it checks if the execution queue is empty or not. If it is empty, the file manager marks the lock as being free. Otherwise, it marks the task instance in the front of the execution queue as having the lock, and sends to this task instance a *grant-lock* message from this file copy.

```
Unlock Write Copy(Task_Instance  Tjk, File  F, Host  I)
{
   A_Manager = locate_public_manager(F);
   if ((A_Manager->Status == ACTIVE) OR (A_Manager->Status == PARTIAL)) {
      if (A_Manager->Execution is empty) {
         A_Manager->Holds_Lock = NULL; }
      else {
         A_Manager->Holds_Lock = front of A_Manager->Execution;
         send_player_grant_lock(A_Manager->Holds_Lock); } }
}
```

Algorithm 4-27: Write Unlock File Copy

Write Release File Copy. To handle the release request by a task instance T_{jk} at site I this service is provided by the local public file manager. After removing T_{jk} from the either the waiting, candidate or execution queues, the file manager checks if the lock needs to be granted to another task instance. If no lock is granted, it then moves task instances from the waiting queue into the candidate queue. The algorithm 4-28 handles the releasing of a public copy of file F by task instance T_{jk} at site I. The FS server first locates the public file manager at the site for the file requested. If the manager is active or partial, it finds the task instance in either the execution, candidate or waiting queues, and removes it from the corresponding queue. The algorithm then checks if the highest priority task instance at this file manager is queued in the waiting queue. If so, the algorithm removes the task instance from the waiting queue and places it into the candidate queue and responds with a *ready* message. If no task instance currently holds the lock on the public copy, the file manager will check if the execution queue is empty or not. If the execution queue is empty, the file manager moves all task from the candidate queue into the execution queue. Otherwise, it grants the lock to the task instance in the front of the execution queue, and sends a response (*grant-lock* messages) to the task instance that now holds the lock.

```
Release Write Copy(Task_Instance  Tjk, File  F, Host  I)
{
   A_Manager = locate_public_manager(F);
   if ((A_Manager->Status == ACTIVE) OR (A_Manager->Status == PARTIAL)) {
      if (find_player(Tjk,A_Manager->Execution) == FOUND) {
         dequeue_player_request(Tjk,I,A_Manager->Execution); }
      else {
         if (find_player(Tjk,A_Manager->Candidate) == FOUND) {
            dequeue_player_request(Tjk,I,A_Manager->Candidate); }
         else {
            dequeue_player_request(Tjk,I,A_Manager->Waiting); } }
      while (find_player(A_Manager->Highest,A_Manager->Waiting)==FOUND) {
         dequeue_player_request(A_Manager->Highest,A_Manager->Waiting);
         queue_player_request(A_Manager->Highest,A_Manager->Candidate);
         send_player_ready(A_Manager->Highest); }
      if (A_Manager->Holds_Lock == NULL) {
         if (A_Manager->Execution is empty) {
            while (A_Manager->Waiting is not empty) {
               Temp = front of A_Manager->Candidate;
               dequeue_player_request(Temp,A_Manager->Waiting);
               queue_player_request(Temp,A_Manager->Candidate);
               send_player_ready(A_Manager->Highest); } }
         else {
            A_Manager->Holds_Lock = front of A_Manager->Execution;
            send_player_grant_lock(A_Manager->Holds_Lock); } } }
}
```

Algorithm 4-28: Write Release File Copy

Write Refresh File Copy. It may be possible, during the creation process of a new public copy file manager at site I, that a request for access by a task instance T_{jk} (local/remote) may have not been handled by the local FS server (the file had not been set to partial yet). Therefore, the FS server requires that when FS clients learn of a new public copy that the FS client makes a request to refresh the information contained at a site. This process ensures the consistency of the information contained in the queues of the file manager. The FS client includes in this refresh request the current state of T_{jk}'s request at all other public copies. In algorithm 4-29, the FS server first locates the public file manager at the site for file F being requested to be refreshed by task instance T_{jk}. If the manager is active or

partial, it queues the task instance into either the execution, or candidate queues depending on the status (allocating/locking) of T_{jk}. A task instance with the status of locking is queued into the execution queue, while all others are queued into the candidate queue.

```
Refresh Write Copy(Task_Instance  Tjk, File  F, Host  I, Request_Status  Status)
{
   A_Manager = locate_public_manager(F);
   if ((A_Manager->Status == ACTIVE) OR (A_Manager->Status == PARTIAL)) {
      if (Status == REQUESTING_LOCK) {
         queue_player_request(Tjk,I,A_Manager->Execution); }
      else {
         queue_player_request(Tjk,I,A_Manager->Candidate); } }
}
```

Algorithm 4-29: Write Refresh File Copy

4.2.2.3 File Server Server Implementation of Physical Access

The FS server provides two services to FS clients (local and remote) the ability to handle RPCs for the writing/reading from a copy located at the site. After handling the RPC both services then check to determine whether to try to delete (reduce the overhead at the site) the copy accessed (see chapter 6).

Handle Physical Write. This service provides the FS server with the ability to handle all remote procedure calls made by a task instance T_{jk} at site I to update the public and shadow copies of file F located at the site. This service also marks the private copy (if located at the site and either active or partial) that a remote update operation has occurred, and therefore needs to be eventually refreshed. The algorithm 4-30 first determines whether there is a local public copy. If there is a public copy, then it checks if the public copy is active or partial. For active copies, the algorithm proceeds to execute the remote procedure call contained in the command message. Otherwise, the algorithm queues the command message into a delayed update queue. This delayed update queue is required, since the entire contents of the physical data is not present at the site. A delayed update is applied when the complete contents of the file has been received. After either completing or delaying the update, the algorithm responds to T_{jk} that the operation has completed. The time that the RPC completed is included in the message. This completion time is used by the remote site to determine whether the update was successful. If no public copy is located at the site, but a private copy is, then the algorithm marks the private copy as having received an update operation and will eventually need to be refreshed.

```
Handle Physical Write(Task_Instance Tjk, File  F, Command_T Command)
{
   if (Is Public Copy local for F) {
      if (Is Public Copy Active) {
         execute_update_operation(Command,F); }
      else {
         delay_update_operation(Command,F); }
      respond_update_done(Tjk,Local_Time); }
   else {
      if (Is Private Copy local for F) {
         mark_update_private(F->Private_Manager); } }
}
```

Algorithm 4-30: Handle Physical Write

Handle Physical Read. The ability to handle all remote procedure calls by task instance T_{jk} to read either the shadow or private copy located at the site is provided by this service. The algorithm 4-31 first determines whether there is a shadow copy located at the local site. If there is, then it proceeds to execute the remote procedure call made by task instance T_{jk} contained in the command message on the contents of the shadow copy. Otherwise, the algorithm executes the remote procedure call on the contents of the private copy. After completing the read operation, the algorithm responds to T_{jk} by sending a response message that the read operation was completed. The time that the remote procedure call completed at is included in this message.

```
Handle Physical Read(Task_Instance  Tjk, File  F, Command_T  Command)
{
   if (Is Shadow Copy local for F) {
      execute_read_operation(Command,F->Shadow_Manager); }
   else {
      execute_read_operation(Command,F->Private_Manager); }
   respond_read_done(Tjk,Local_Time);
}
```

Algorithm 4-31: Handle Physical Read

4.2.2.4 File Server Server Implementation of File History

The FS server is required to maintain history on the last five read/write requests made by a task T_j that occur on a file F (either accessed by or located at the local site). This information then can be used by the File Assigner server to determine when and how to change the distribution of copies of files in the MELODY system. The File Server/Task Scheduler use this information to determine when and how to invoke the File Assigner. The FS server provides two services for the updating the file history information. Access to the file history information itself is provided by allowing the File Assigner, Task Scheduler and File Server to directly access the file history. As a result, no service calls are provided.

Update Write Access. This service provides the FS server with the ability to update the file history that a write has occurred to a public copy located at the site. This write is characterized as successful or failed, and local or remote. Both local and remote write requests update the global write history, but only local write requests update the local write history. The algorithm 4-32 first determines if the write request was local or remote. If remote, then nothing is done. Otherwise, the algorithm checks if the access was successful. If successful, the algorithm updates the local write file history information with the successful write request. Otherwise, the local write file history is updated with a failed write request. After updating the local write access history the algorithm updates the global write history. This is done by checking again if the write was successful. If successful, the algorithm updates the global write file history with a successful write request. Otherwise, the algorithm updates the global write file history with a failed write request.

```
Update Write Access(Status_T  The_Status, Location_T  The_Location,
                    File_History_T  The_History)
{
   if (The_Location == LOCAL_HOST) {
      if (The_Status == SUCCESSFUL) {
         update_history_success(The_History->Write_Local); }
      else {
         update_history_failed(The_History->Write_Local); } }
   else { }
   if (The_Status == SUCCESSFUL) {
      update_history_success(The_History->Write_Global); }
   else {
      update_history_failed(The_History->Write_Global); }
}
```

Algorithm 4-32: Update Write Access

Update Read Access. Updating the file history information with the information that a read access has occurred is provided by this service. This read access is characterized as either successful or failed, sensitive or robust, and the location (local/remote) where the access occurred. The algorithm 4-33 first looks to determine if the read access is local or remote. If remote, then nothing is done. Otherwise, the algorithm checks if the access was considered robust. If robust, the algorithm checks if the access was successful or failed. If successful, the algorithm updates the read robust file history with the successful read request. Otherwise, the read robust file history is updated with a failed read request. For sensitive and essentially sensitive access, the algorithm checks if the access was successful. If successful, the algorithm updates the read sensitive file history with the successful read request. Otherwise, the read sensitive file history is updated with a failed read request.

```
Update Read Access(Status_T  The_Status, Sensitivity_T  The_Sensitivity,
                   Location_T  The_Location, File_History_T  The_History)
{
   if (The_Location == LOCAL_HOST) {
      if (The_Sensitivity == ROBUST) {
         if (The_Status == SUCCESSFUL) {
            update_history_success(The_History->Read_Robust); }
         else {
            update_history_failed(The_History->Read_Robust); } }
      else {
         if (The_Status == SUCCESSFUL) {
            update_history_success(The_History->Read_Sensitive); }
         else {
            update_history_failed(The_History->Read_Sensitive); } }
}
```

Algorithm 4-33: Update Read Access

4.2.2.5 File Server Server Implementation of File Movement Handler

The File Movement sub-server of the FS server handles the physical data transfer of physical files to/from one site from/to another site within the MELODY system. The sender and receiver services of the file movement handler accomplish this service of transferring a data file between two sites. Both modules use TCP connections to guarantee that no data loss is experienced by the servers (underlying medium data loss may exist, but is handled at that layer). Details can be found in appendix A regarding the MELODY file transfer communication model.

Sending Service. The *sender* receives a request to send a copy of a file located on the local site to the remote site making the request. The sender then proceeds to complete the following steps to satisfy that request for a file by the site:

1: **Connect**. The sender connects to the receiver located at site I.
2: **Open**. The sender opens the requested file copy (this copy is a duplicate of the public copy such that no additional disturbance is caused on the public copy).
3: **Header**. A file header message containing the file identifier and length is sent to the receiver at site.
4: **Data**. The complete contents of the file copy is sent by the sender to the receiver. This is done by having MELODY send data portions of the file (size based on communication medium) until the complete contents (length L) of the file copy F has been sent.
5: **Disconnect**. The sender disconnects from the receiver.
6: **Close**. The sender closes the requested file copy F.

Receive Service. The *receiver* begins the process by accepting a connection request by a remote *sender* located at a site which begins the receive operation for a file copy. The receiver then completes the receive operation by completing the following steps to receive the file sent by the site:

1: **Accept**. The sender accepts a connection request from a sender located at site I.
2: **Header**. A file header message is received and decoded by the receiver. This header contains the file identifier F and the length L of the file to be received.
3: **Create**. The receiver creates a new file copy of length L.
4: **Data**. The contents of the file are received by the receiver from the sender and written into the newly created file copy (continues until the entire length of the file has been received).
5: **Disconnect**. The receiver disconnects with the sender.
6: **Close**. The receiver closes the file copy.
7: **Acknowledge**. The receiver acknowledges to the local File Assigner that the file has been received at the site.

Chapter 5 Task Scheduler

Tasks in MELODY represent control functions, corrective actions etc., and are executed on a regular basis. Tasks are also viewed as executing on dedicated processors with file access being done using remote file operations rather than task migration or file transfer to the accessing site (see section 3.1). Each task is a small-scale, transaction-like operation with just one segment of task activity termed *critical section* in which it accesses a number of objects (copies of possibly different files) concurrently. Write operations work on all copies of a file, through remote procedure calls, while read operations read from only one copy. It is also assumed that the execution times on each local file copy could be determined within tight bounds.

5.1 Task Scheduler Model

In MELODY, the principle has been established to reverse the order of task and resource scheduling (see section 1.2) in order to abort tasks as early as possible and at the same time lock resources as late as possible. This reversal, however, leaves the Task Scheduler without accurate information on task execution times (the actual resource allocation time is unknown when the Task Scheduler is invoked) and a task instance would then have to be scheduled based on estimates. The Task Scheduler is invoked by the Run-Time Monitor (see section 1.2. Details can be found in chapter 7). The Task Scheduler also collects information which could then be utilized to determine when and for which files the File Assigner is invoked. The Task Scheduler cooperates with the local File Server and File Assigner, but provides for no cooperation with MELODY modules at remote sites.

The Task Scheduler schedules a set of k tasks that have arrived at a site I according to a scheduling policy tailored to the applications needs. In the current implementation of MELODY this earliest deadline first scheduling algorithm is used by the Task Scheduler to schedule a set of task instances. A task instance T_{jk} would be considered for scheduling using: EET_{jk} (estimated execution time), DT_{jk} (deadline), LRF_{jk} (list of required files) and PrT_j (static priority). When invoked, the Task Scheduler utilizes a service provided by the local File Server to determine EET_{jk} for those task instances T_{jk} which have arrived in the meantime. The service calculates the EET_{jk} (see section 3.4) which is based on the estimated location and acquisition times (derived from the average of the corresponding values of the last five instances of T_j) and the computation time (time required to access the needed files and their copies, including the communication time for the possibly remote operations).

The Task Scheduler then schedules task instances ordered by their static priority and deadline. Task instances are placed into the *local task queue* (***LTQ***) according to their order of execution, and dispatched from execution from this queue. Task instances that can be scheduled by the Task Scheduler are termed as ***strongly schedulable***, and are designed to be placed into the *strongly schedulable queue* (***SSQ***). The Task Scheduler also may build a queue of task instances (termed ***weakly schedulable***) that could have been scheduled if copies of files currently located at another site where locally available or if there had been fewer public copies of the files needed by the task instance. Based on the relative degree of sensitivity of the task instance, weakly schedulable task instances are separated into the following three categories:

Write: This category includes all task instances that would require at least one public copy be deleted to be scheduled into the SSQ. These task instances are termed **weakly schedulable write** and are placed into the weakly schedulable write queue (WWQ);

Sensitive: Includes all task instances requiring the creation of a public (and a shadow) copy at the local site to be scheduled into the SSQ. These task instances are termed **weakly schedulable sensitive** and are placed into the weakly schedulable sensitive queue (WSQ);

Robust: Tasks that could be scheduled into the SSQ if a private copy was locally available are termed **weakly schedulable robust** and placed in the weakly schedulable robust queue (WRQ). Whether the task instance could also utilize a private copy depends on whether the task instance is considered robust based on the values of b_i" (robust threshold) and R_{jk} (relative degree of sensitivity) used by definitions 3 and 4 (see section 1.3).

All task instances that are not placed into SSQ, WWQ, WSQ or WRQ would be termed **non-schedulable** and placed into the non-schedulable queue (NSQ). Essentially critical task instances are not handled by the Task Scheduler, but are immediately given to the File Server by the Run-Time Monitor (see chapter 7). Once the Task Scheduler has determined which task instances are *strongly schedulable* and *weakly schedulable*, it sends all strongly schedulable task instances to the File Server to begin their location phase. All non-schedulable task instances would then be aborted. Due to the unpredictable environment typical of safety-critical system, which results in the ensuing typical aperiodicity of task instance occurrences in that environment, and the fact that task scheduling is very short in duration, task scheduling in MELODY is non-preemptive.

Using the information regarding weakly schedulable task instances the Task Scheduler could then invoke the File Assigner to change the distribution of public copies in order to try to improve the schedulability of these tasks. For example, if the number of task instances queued in *WSQ* surpassed a predetermined threshold value the Task Scheduler would then request that the File Assigner create a local public copy of this file in order to provide faster local access to this file. For the tasks queued in *WWQ* the Task Scheduler would request that the File Assigner reduce the number of public copies in the system in order to reduce the time required to access the remaining copies. The Task Scheduler would request for the tasks queued in the *WRQ* that a private copy be created since this would improve their performance while not significantly impacting the performance of writing tasks within the system. Details regarding this type of File Assigner integration (**Task Scheduler-Oriented Integration**) will be described in section 6.1.4 as one of two distinctly different integration methods.

5.2 Task Scheduler Implementation

As a summary of the presentation above the Task Scheduler is presented in pseudo code in algorithm 5-1. The algorithm begins going through the set of k task instances (task instances which are waiting for the TS) in the order based on their static priority. For every T_{jk}, TS begins by determine the EET_{jk} for every task instance T_{jk}. The algorithm then tries to schedule T_{jk} into the local task queue (LTQ) (based on the earliest deadline first scheduling algorithm) using the service schedule_task_instance. This service determines if T_{jk} can be scheduled into the LTQ without causing any task instance already in the LTQ to not be schedulable. If successful, the algorithm schedules T_{jk} into the LTQ and queues T_{jk} into the SSQ. If T_{jk} could not be scheduled based on T_{jk}'s current EET_{jk}, then TS adjusts T_{jk}'s EET_{jk} by calling the service local_execution_time. This service returns the execution time for T_{jk} based on the assumption that all files are locally available. Using T_{jk}'s adjusted EET_{jk},TS tries again to schedule T_{jk} into the LTQ. If successful, then it queues T_{jk} into one of the three WWQ, WSQ or WRQ, based on T_{jk}'s R_{jk}. Finally, if TS could not schedule T_{jk} using T_{jk}'s adjusted EET_{jk}, then TS queues T_{jk} into the NSQ.

```
Run_Task_Scheduler(Waiting_Task_Instances   k)
{
   for every Tjk in k ordered by Priority and Deadline {
      Tjk->EET = Estimated_Execution_Time(Tjk);
      if (schedule_task_instance(Tjk,LTQ) == SUCCESS) {
         queue_schedulable_task(Tjk,LTQ);
         queue_instance(Tjk,SSQ); }
      else {
         Tjk->EET = Local_Execution_Time(Tjk);
         if (schedule_task_instance(Tjk,LTQ) == SUCCESS) {
            if (needs_private_copy(Tjk) == SUCCESS) {
               queue_instance(Tjk,WRQ); }
            else {
               if (needs_shadow_copy(Tjk) == SUCCESS) {
                  queue_instance(Tjk,WSQ); }
               else {
                  queue_instance(Tjk,WWQ); } } }
         else {
            queue_instance(Tjk,NSQ); } }
   if (Task_Scheduler_Oriented_Integration) {
      request_file_distribution_changes(WWQ,WSQ,WRQ); }
   for every Tjk in SSQ {
      Begin_Location_Phase(Tjk); }
   for every Tjk in WRQ, WSQ, WWQ and NSQ {
      Begin_Abort_Task(Tjk); }
}
```

Algorithm 5-1: Run Task Scheduler

Once the algorithm has tried to schedule all waiting task instances, it proceeds to determine is the Task Scheduler-Oriented Integration method is being used. If so, then the Task Scheduler will request the file distribution to be changed based in the task queued in the three weakly schedulable task queues: WWQ, WSQ and WRQ. After this the algorithm would for every task instance that has been queued into the SSQ call the FS client service begin location phase. For every other task instance the algorithm calls the service begin_abort_task to complete the abortion of the task instance and remove it from the MELODY system.

Chapter 6 File Assigner

In MELODY, a trade-off has to be made, under changing requests and deadline failure patterns, between the costs of serving file requests with a given distribution of public copies, and the costs for realizing various alternative distributions. The term cost here denotes time delays for overhead operations, for communication and transmission delays (both local and remote). A large number of public copies are advantageous for reading task instances, since such a public copy is more likely to be locally available. This local copy allows for faster access since there would be no remote communication required and therefore no inherent delays in accessing the copy. However, writing task instances suffer from this increased number of copies. The increased cost for updating a public copy grows as the number of public copies increases due to the additional communication and local processing required to handle responses from additional sites (even though the updating could be performed relatively in parallel at the remote sites). In MELODY, the File Assigner (FA) is responsible for adaptively changing the distribution of public file copies according to the changing and typically unpredictable environment. The File Assigner then uses information obtained from the local site and from remote sites to determine which of the alternatives should be taken to improve the survivability and performance of the local site and the MELODY system. The alternatives for public copies are:

Relocation. Relocating of a public copy from a remote site to the local site;

Replication. Creating an additional public copy at the local site;

Deletion. Deleting a public copy if there were not enough requests over a period of time, or under emergency (nearly essentially critical task failure) conditions.

The local File Assigners cooperate with the local File Server or local Task Scheduler (depending on two different methods of File Assigner integration, to be described in section 6.1.4) and with the remote File Assigners to manage replication, relocation and deletion of public copies within MELODY. Private copies are not managed by the File Assigner since decisions for creation or deletion are solely local and require no consensus from remote sites (see section 3.4).

6.1 File Assigner Model

The File Server module uses a client/server approach to handling requests for the relocation, replication and deletion of files. This allows the File Assigner client to concentrate on satisfying requests made by the local MELODY servers, while the File Assigner server concentrates on handling requests for information from File Assigner clients (local/remote). The division of responsibilities between the File Assigner server and client has then been subdivided in the following way:

Client: The client is responsible for requesting information regarding the distribution of files from the File Assigner servers. Deciding how (based on information received) best to change the file distribution to improve cost of accessing files not only from the local site, but also in respect to the needs or costs that a change would have on remote sites. It also ensures that the request for an authorized change in the file distribution was completed by requesting an acknowledgment from the File Servers;

Server: The server handles all requests for information regarding their site's costs or need for copies of files located at its site. Since the replication, relocation and deletion of public copies can drastically effect the performance on remote sites, the File Assigner clients must retrieve an FA Lock on the file. This ensures that only one File Assigner client changes the distribution of public copies at any time (see section 6.1.1).

This partition of the File Assigner module can be seen in figure 6-1. As a result of this distribution of responsibilities, the File Assigner client provides no services to remote sites, while the File Assigner server provides all required information to the local/remote File Assigner clients. Details regarding the integration of the File Assigner can be found in section 6.1.4.

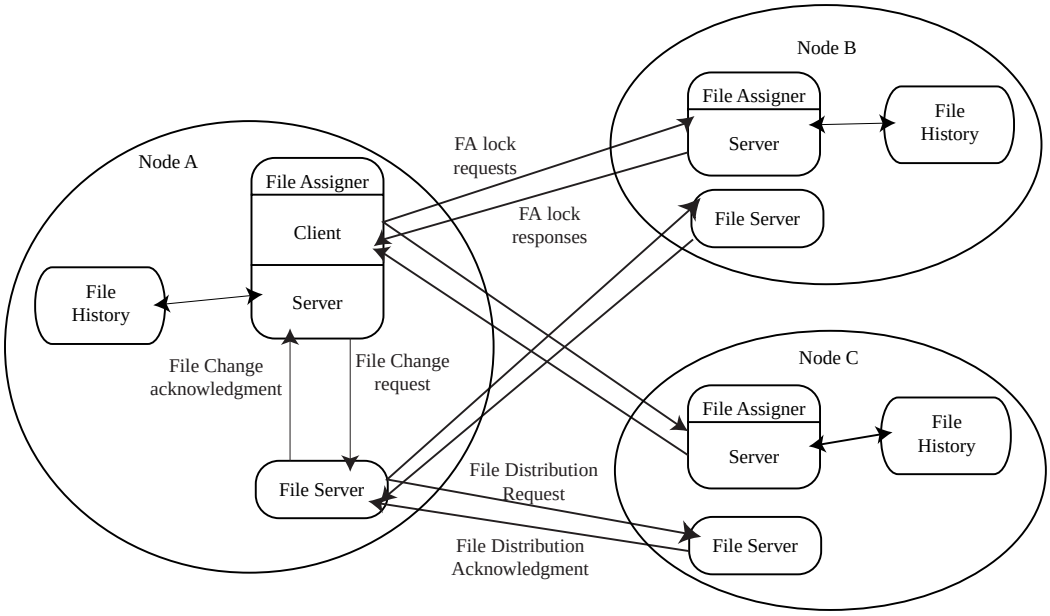


Figure 6-1: File Assigner Client/Server Model

6.1.1 File Assigner Lock Protocol

The ***File Assigner lock*** procedure (*FA Lock*) ensures that at any point of time only one File Assigner is allowed to change the distribution of public copies of a file. Without this control, it would be possible that two (or more) File Assigner could at the same time request to delete two different public copies of a file (without knowledge that the other had also requested a delete). This might violate the requirement for maintaining a minimum number of copies (see section 3.2). A second reason to maintain a FA Lock is to ensure that only one additional copy is created at any time. Since the number of public copies has a significant effect on the performance of write operations, in MELODY the number of public copies will only be changed by one within a given interval of time. In this way a change would be reflected in the file history information which could then be utilized more accurately in determining whether to create additional copies. In summary, the File Assigner only changes the distribution of public copies when it has been granted the FA Lock on a given file. Once it has been granted locks on all public copies of a file it may then proceed to request to change the distribution of public copies. After the change has been acknowledged it will release the FA Lock.

The protocol used to manage the FA Lock is the **Priority Insertion protocol** that has been defined by D.C. Daniels [Dan92, WeD91a, WeD91b, and WeD94]. The Priority Insertion protocol is designed to minimize *priority inversion*. To accomplish this, each Priority Insertion scheduler (File Assigner server), which controls access to a file, maintains a schedule ordered by priority. When a client (File Assigner client) needs to change the distribution of a file, it sends an access request message to the scheduler at each site holding a public copy of the file. Upon receipt of the client's message, the scheduler insert this request into the local schedule. If no other client holds the FA Lock for this file, then the scheduler will send a lock grant message to the client. The Priority Insertion protocol utilizes the total order over clients (ordered by unique server-id and host-id) to maintain schedules such that between any two schedules, there is agreement on the partial ordering of common clients. Provided that a client at the front of a schedule always holds the FA Lock for the file, the Priority Insertion protocol is deadlock free. However, there is no guarantee that at any given time a client at the front of the schedule holds the FA Lock for the file (a lower priority client may already have been granted the FA Lock at this location). Moreover, this ambiguity can lead to disagreement between the schedulers which may lead to deadlock. To resolve this condition, a Call Back Mechanism is utilized (see section 4.1.1).

6.1.1.1 File Assigner Client Operations

The operations used by a client trying to obtain the FA Lock are basically sequential and will be described as a step-wise procedure.

Step 1: The client begins to obtain the FA Lock for a file by sending an access request message to schedulers at all locations holding a public copy of the file.

Step 2: The client now waits while the public file copies respond with lock grant messages. As each lock is granted, the sending scheduler, is added to a set of locked sites. If all sites have responded with granted FA Lock, the client will proceed to step 3. The client will acknowledge all call back messages while in step 2 (ignored in all other steps). For a description of call back messages and their purpose, see the Call Back Mechanism defined in section 4.1.1.

Step 3: The client begins to change the distribution of public copies (all needed locks have been obtained).

Step 4: Upon completion, the client sends release messages to all scheduler holding a public copy of the file.

6.1.1.2 File Assigner Server Operations

The operations handled by the scheduler for the Priority Insertion protocol are completely message driven. Thus, the behavior of the scheduler can be expressed as message handlers which define the actions taken in response to incoming messages.The variable HOLDS_LOCK indicates which client that holds the FA Lock for the public copy. Procedures common to all request handlers are: SEND (sends a message to a client); INSERT (inserts a requesting client into priority ordered queue *Q*); REMOVE (removes a requesting client from queue *Q*).

Access Request Handler. Receiving an *access* message from a client, causes the scheduler to execute the following algorithm. If the FA Lock queue is empty the scheduler inserts the client into the FA Lock queue, sets HOLDS_LOCK to this client, and sends a lock grant message to the requesting client. Otherwise, it inserts the client into the FA Lock queue. It then determines if the requesting client is in the front of the FA Lock queue and the HOLDS_LOCK variable is set to another client. If so, the scheduler will request to call-back the FA Lock from the lower priority (lower order server-id and host-id) client that currently holds the FA Lock and sets the HOLDS_LOCK variable to a free status.

```
file_assigner_request_access(file assigner client FAC_i)
   {
      if (FA Lock queue == NULL) {/* is the FA Lock queue empty */
         INSERT(FAC_i,FA Lock queue);
         HOLDS_LOCK = FAC_i;
         SEND("lock grant",FAC_i); }
      else {
         INSERT(FAC_i,FA Lock queue);
         if (front(FA Lock queue) == FAC_i) { /* FAC at the front of FA Lock queue */
            if  (HOLDS_LOCK == NULL) { /* is the lock free */
               SEND("call back",HOLD_LOCK);
               HOLDS_LOCK = NULL; } } }
   }
```

Algorithm 6-1: Priority Insertion Access Request Handler

Release Handler. When the scheduler receives a release message from a client, it executes the following algorithm. The algorithm first removes the client from the FA Lock queue, and sets the HOLDS_LOCK variable to a free status. It then checks if their is another client queued. If so, the algorithm then grants the FA Lock to the client in the front of the FA Lock queue and sets the variable HOLDS_LOCK to this client.

```
file_assigner_request_release(file assigner client FAC_i)
   {
      REMOVE(FAC_i,FA Lock queue);
      HOLDS_LOCK = NULL;
      if (FA Lock queue == NULL) {/* is the FA Lock queue empty */
         HOLDS_LOCK = front(FA Lock queue);
         SEND("lock grant",front(FA Lock queue)); }
   }
```

Algorithm 6-2: Priority Insertion Release Handler

Call Back Acknowledgment Handler. When a call back acknowledge message is received by the scheduler it executes the following algorithm. It sends a lock grant message to the client in the front of the FA Lock queue and sets the variable HOLDS_LOCK to this client.

```
file_assigner_acknowledge_callback(file assigner client FAC_i)
   {
      HOLDS_LOCK = front(FA Lock queue);
      SEND("lock grant",front(FA Lock queue));
   }
```

Algorithm 6-3: Priority Insertion Call Back Acknowledgment Handler

6.1.2 File Assigner Client Functionality

In MELODY, the FA client is in charge of handling requests to change the distribution of public copies in order to try to improve the performance of tasks at the local site. However, any change effects the performance of tasks at remote sites. Therefore, a request to create, move or delete a copy would only be done if the FA servers, at the sites involved, would allow the change to occur. The FA servers make the decision to grant or deny a request based on its effect on the tasks located at their sites (this will be described in section 6.1.3 as part of the FA server functionality). For example, the creation of an additional public copy would improve the performance of the local read tasks, however it could be detrimental to write tasks within the system due to the additional overhead required to access an increased number of copies. Deletion of a copy improves the performance of write tasks within the system, however it causes read task at that site to access a remote copy. This increases their access times making it more difficult for them to complete prior to their deadlines. The FA client provides four services which could be invoked under various situations to request changing the distribution of public copies, but only after obtaining a consensus amongst all FA servers involved. Under the situation where a large number of sensitive local read tasks are failing, the FA client requests to create a local public copy (by either relocating or replicating a copy from a remote site) using the service ***get local public copy***. As stated above, this improves the read access times of local tasks. A remote FA server would grant relocation of its copy if it would not be detrimental to its read tasks. It grants replication of an additional public copy only if it would not cause significant problems for its local write tasks. Where an excessive number of write tasks are failing, the FA client requests to reduce the number of public copies using the service ***reduce number of public copies***. Remember, this is done to reduce the time required by future write tasks to access the remaining copies. However, a copy (somewhere within the system) would only be deleted if it would not significantly impact the performance of read tasks at the site where the copy resides (the FA server at that site granted the deletion of its copy). After a period of time, there may no longer be a high enough number of local sensitive read tasks to justify having a public copy locally available. When the local File Server detects this it invokes the local FA client. This module then executes ***delete local public copy*** in order to reduce the overhead of maintaining the public file copy in question. It seeks an FA consensus to delete the copy unless the requirement to maintain a minimum number of public copies is violated. Following the detection of a nearly essentially critical write task failure, it's necessary to request to delete a public copy, somewhere within the system, in order to improve the chances that the upcoming essentially critical task will complete prior to its deadline. The failure would be detected by either the File Server or Task Scheduler (see section 6.1.4). Using the service ***emergency reduction of public copies***, the FA client requests to delete a public copy regardless of whether or not the remote sites would allow it based on its local read tasks (as in the *reduce number of public copies* service).

To determine when to get or delete a public copy the FA client uses the file history information

(maintained by the File Server described in section 4.1.5). This history information records the number of task instances which have requested a certain type of access to a file copy within a predetermined time interval (set upon system start-up and conceived to be adaptable, over certain time intervals, to specifics of the task profiles encountered). Thus the information contained in the file history represents a set of task instances that have either succeeded or failed during a fixed window of time. The FA client then requests a file distribution change based on the following preset file threshold values:

Get Copy Threshold:When the number of failed read (both sensitive and robust) access requests for a file surpasses this threshold the FA client sends a request to receive a public copy at its site by either relocation or replication. If at least one FA server grants the relocation of its public copy, then its copy will be moved to the local site. Otherwise, the requesting FA client checks if a copy could be created. If all FA servers granted the replication of a public copy, then a new public copy would be created, otherwise no copy would be create or moved.

Reduce Copy Threshold:Once the number of failed write requests for a file surpasses this threshold the FA client sends a delete request to all FA servers holding a public copy of the file. If at least one of the FA servers grants the deletion of its copy, then the public copy at the location with the lowest number of local read access requests will be deleted. If all FA servers denied the deleting their copy, then no copy is deleted.

Note again that the FA client does not handle requests for the creation or deletion of local private copies since such a decision requires no consensus amongst the remote sites. When there are no longer enough read access requests the local public copy will be deleted by the File Server.

To ensure that a change in the distribution of a file does not occur to frequently a request is only made if the time between this request and the prior request to change the distribution of this file has exceeded a **change waiting time** value. In this way only one change to the distribution of a file occurs within a predetermined interval of time. This value is set upon system start-up. In the future it is conceived to be adaptable, over certain time intervals, to the task profiles encountered.

In summary, the protocols used to implement the services of the FA client are described in the following paragraphs.

Get Local Public Copy. Once the local FA client has determined that (based on the number of failing local read access requests and the *get copy threshold*) a public copy should be created at the local site it executes this service (protocol shown in figure 6-2). The FA client sends *get copy* requests to all FA servers at sites holding a public copy. These, in turn, would grant or deny an FA lock (see section 6.1.1). If the lock has been granted the corresponding FA server includes in its response the information regarding whether the remote site would grant the relocation or replication of an additional public copy to the local site (based on the remote FA server's need). If any site denies granting the FA Lock the requesting site will release all previously obtained FA Locks and mark the request as finished. The remote FA server grants relocation of its local public copy if the local read access requests to this copy would not significantly suffer from accessing a remote public copy. Replication is granted by the remote FA server if the local write access requests would not significantly suffer from an additional copy. Specifically how the remote FA servers grant or deny a request will be described as part of the FA server functionality (see section 6.1.3). After receiving responses from all remote FA servers, the requesting FA client first determines if a remote site would allow it to move its public copy to the local site. If so, the FA client issues a request to the local File Server to move the copy from the granting site to the local site. If no site permitted the relocation of a copy, then the FA client checks if all sites agreed to create a public copy. If creation is possible, then the local FA client issues a request to the local File Server to create a new public copy at the site. If none of the measures are approved, the requesting FA client will do nothing. At the end of this service the FA client releases all FA Locks that it had obtained in the beginning of the service.

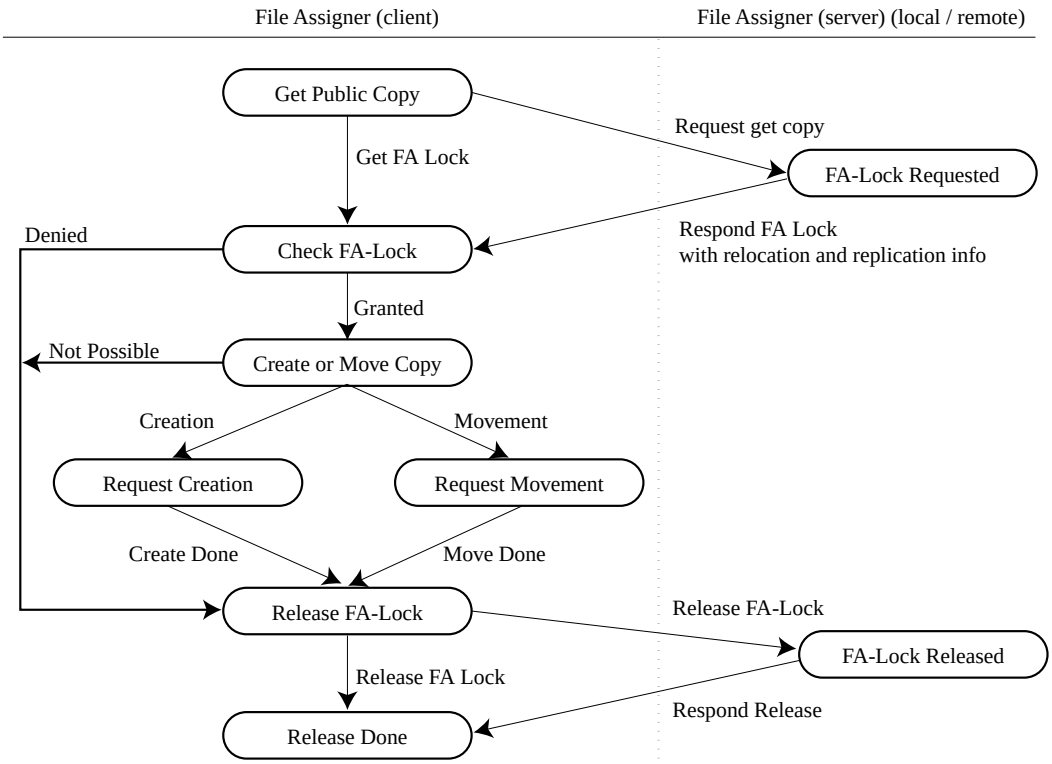


Figure 6-2: Get Local Public Copy Protocol

Reduce Number of Public Copies. The local FA client executes this service (protocol shown in figure 6-3) after determining that (due to the number of writing task instances failing and the *reduce copy threshold*) the number of public copies in the system should be reduced. *Delete copy* requests are sent to all FA servers at sites holding a public copy. In response, they would grant or deny an FA lock (see section 6.1.1). If a lock has been granted the corresponding FA server includes in its response the information regarding whether the remote site grants or denies the deletion of its local public copy is based on the remote FA server's need (see section 6.1.3). If any site denies granting the FA Lock the requesting site releases all previously obtained FA Locks and marks the request as finished. After all sites have responded, the FA client checks if at least one FA server granted the deletion of its copy. If so, the FA client issues a request to the local File Server to delete the copy. If deletion is not possible the FA client proceeds to the next step of releasing the FA Lock. At the end of this service the FA client releases all FA Locks that it had obtained in the beginning of the service.

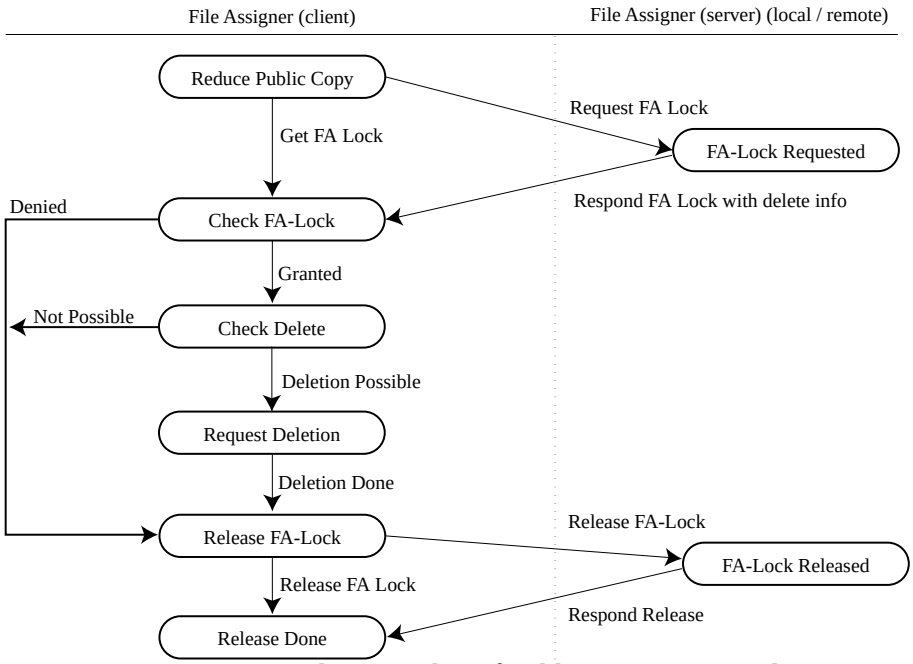


Figure 6-3: Reduce Number of Public Copies Protocol

Delete Local Public Copy. This service (protocol shown in figure 6-4) is executed after the local File Server has determined to delete the local public copy. The difference between this service and the *reduce number of public copies* service is that there is no requirement that the FA client must query the remote FA servers regarding the impact that this deletion would have on their sites since deletion of the local public copy would not effect remote sites. The FA client must only obtain the FA lock. After all sites have responded, the FA client ensures that the deletion of the local public copy does not cause the number of copies to fall below the minimum number (see section 3.2). If this condition is met, the FA client issues a request to the File Server to delete the copy.

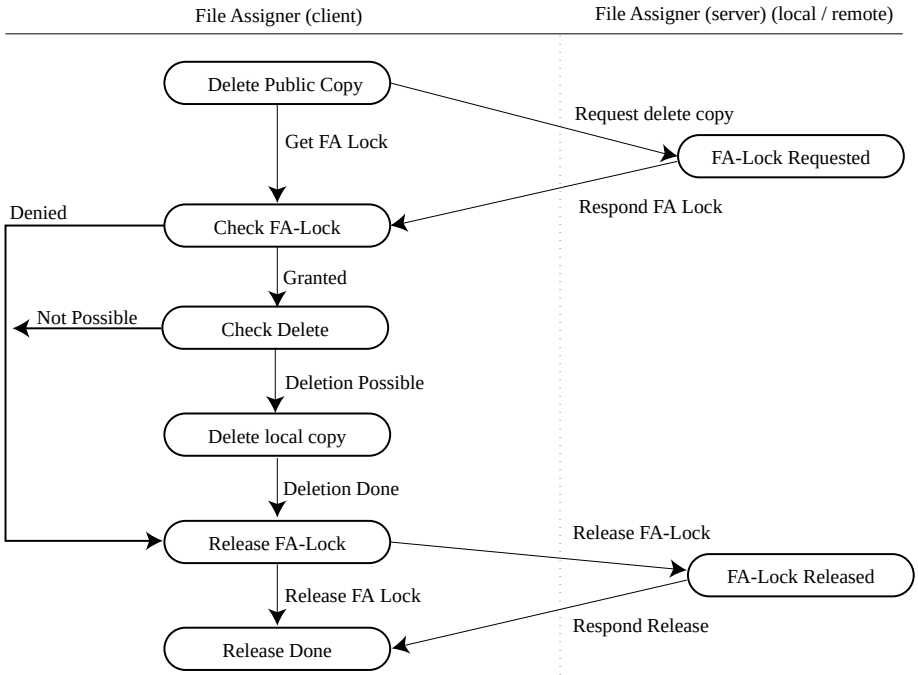


Figure 6-4: Delete Local Public Copy Protocol

Emergency Reduction of Public Copies. Following the failure of a nearly essentially critical write task instance *(an emergency situation since the survivability of the system is in danger)*, this service (protocol shown in figure 6-5) would be executed in order to reduce the number of public copies. This improves the chance that the next invocation of the failing task (which will be essentially

critical) will complete prior to its deadline. Requests are sent to all FA servers at sites holding a public copy. In response, they would grant or deny an FA Lock (see section 6.1.1). If a lock has been granted the corresponding FA server includes in its response the read access file history for its site. If any site denies the FA Lock the requesting site releases all previously obtained FA Locks. After all sites have responded, the FA client chooses to delete the public copy located at the site with the lowest number of read access requests (as long as it would not violate the requirement to maintain a minimum number of copies). It then issues a request to the File Server at that site to delete the copy. If it could not delete a copy the FA client releases the FA locks. This would as well be done after a copy has been successfully deleted. It is conceivable that in the assumed case of emergency the minimum number of copies requirement would be superseded by the need for the whole system to survive. For future implementations and for real applications the emergency service described above could then be tailored in ways in which the reliability requirement is basically ignored.

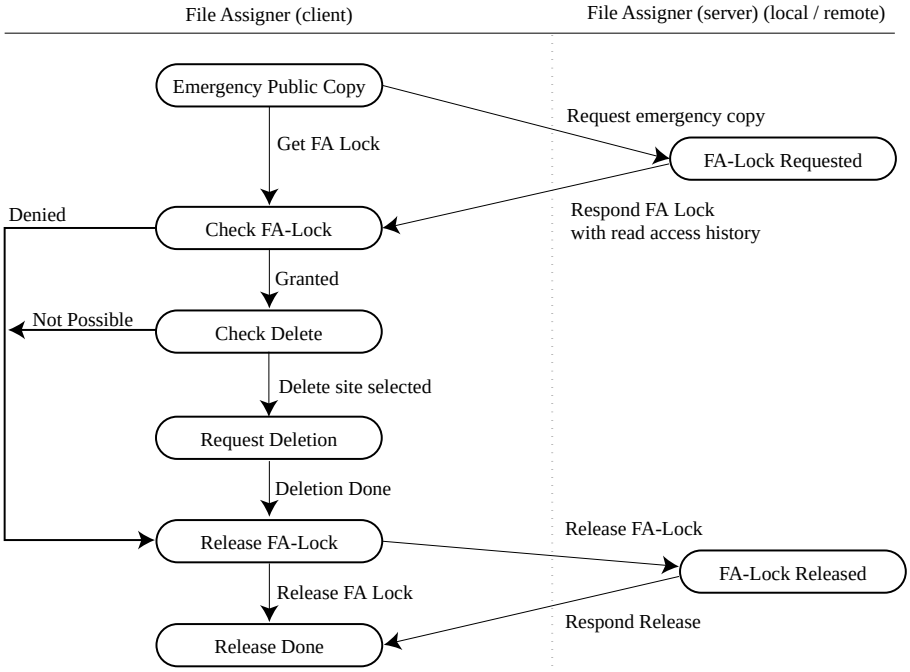


Figure 6-5: Emergency Reduction of Public Copies Protocol

6.1.3 File Assigner Server Functionality

The FA server at a given site handles requests from the FA clients for information regarding the specific needs of the site for the local public copy. Since the creation and deletion of private copies is solely a local decision (a private copy located at one site does not effect another site's performance in a significant way), the FA server handles no requests regarding private copies. The FA client must first receive an FA Lock on the file (described in section 6.1.1). If the FA server grants this lock then it includes with the grant response the information on its local site's needs. If it denies the lock then the FA server does not include any information. For getting a public copy, the FA server includes whether or not it would allow the FA client to either move its copy, or create an additional copy. The FA server allows moving its copy if the number of failed sensitive read task instances in the file history (see section 4.1.5) is less than the ***delete copy threshold***, otherwise it will deny movement of its copy. Creation of an additional public copy is allowed by the FA server, if and only if the number of failed writing task instances in the file history is less than the ***allow copy threshold***, otherwise it will deny replication of a copy. Both values for the *allow copy threshold* and *delete copy threshold* would be set during start-up of the system, and they are conceived to be adaptable to the needs of the local site for this specific file. Note that the decisions to allow replication or relocation of a public copy are mutually exclusive. The FA server grants deleting its local public copy, if the number of failed reading task instances in the file history is less than the *delete copy threshold* and the number of public copies exceeds a preset minimum number of copies (see section 3.2), otherwise it denies deletion of its copy. The FA server then sends the response back to the requesting FA client (the actions taken by the FA

client in response to this message are described in section 6.1.2).

6.1.4 File Assigner Integration

It is expected that invocation of the File Assigner as early as possible would improve the system's survivability since file copies could be relocate, replicated or deleted as early as possible. As a result there would be improved chances that file distribution changes would be completed prior to the arrival of the next task instances. Invoking the File Assigner as late as possible however would allow for a more accurate decision to be made in regards to the actual needs of task instances at a site, but this could impact the system's survivability by delaying the time before a decision is made to change the distribution of copies. Two distinctly different models for the integration of File Assigner were developed: *Task Scheduler-Oriented Integration* and *File Server-Oriented Integration*.

Task Scheduler-Oriented Integration. In this model the Task Scheduler (see section 5.1) maintains a queue of *weakly schedulable sensitive read tasks* (**WSQ**), a queue of *weakly schedulable robust read tasks* (**WRQ**), and a queue of *weakly schedulable (sensitive) write tasks* (**WWQ**). Each of the weakly schedulable tasks is kept in the corresponding queue for a fixed amount of time. Thus the contents of a queue represent a set of tasks that have (weakly) failed during a fixed time window. Upon completion of task scheduling, the Task Scheduler sends requests to the File Assigner based on the following threshold values:

WWQTh: When the number of task instances queued in *weakly schedulable (sensitive) write task queue* (WWQ) surpasses this threshold the Task Scheduler sends requests to the local FA client to reduce the number of public copies for each file required by these tasks. This would have the effect of improving the schedulability of the tasks by reducing the time required to access the public copies of these files.

WSQTh: Once the number of task instances queued in the *weakly schedulable sensitive read task queue* (WSQ) surpasses this threshold the Task Scheduler sends requests to the local FA client to create a local public copy for every file required (by these task instances) that currently is not available at the local site. This improves the schedulability of these tasks by reducing the time required to access these files (since local access time is much faster then remote access with its additional communication overhead).

WRQTh: Once this threshold has been surpassed, the Task Scheduler would request from the File Server to create a local private copy for each file required by the task instances queue in *weakly schedulable robust read task queue* (WRQ). The schedulability of these tasks would then be improved for the same reason regarding local access times stated for WSQTh. Note that the creation of private copies is solely under the control of the File Server since this effects only the local sites performance and therefore there is no requirement to arrive at a consensus among the remote sites on the impact that the new private copy would have.

Note that these threshold values would be set during a initialization phase of the system and would be experimentally tailored to the specific application environment (in the same way that deadlines of the task instance would be adjusted to the application environment).

After a period of time there may no longer be enough local read accesses to justify having a copy (public or private) locally available due to the overhead incurred by maintaining the local copy. The File Server using the *delete copy threshold* (see section 6.1.3) would make its decision whether or not to delete the local copy. The File Server requests to delete the local public copy (using the service *delete public copy)* when the number of read access requests (both sensitive and robust) underpasses this threshold. For private copies the File Server deletes the copy based on whether the number of robust read access requests underpasses this threshold.

In addition, following the abortion of nearly essentially critical task instance the File Server requests for the files required by this task instances that either the number of public copies be reduced (write access), or a local private copies be created (read access). Both actions improve the chance (by reducing the acquisition and execution times) that the next incarnation of the task will succeed. For reading task instances, this tries to ensure that when the next invocation $T_{j(k+1)}$ does occur that private

copies for all read access requests are locally available. This significantly reduces the execution time required by the task instance since the copy would be locally available and would not require any remote communication to access the copy (see section 3.2). For writing task instances, it should be ensured that the number of public copies has been reduced. The reduced number of public copies would in turn reduce the execution time required to access the copies by reducing the overhead caused by communicating with a increased number of copies.

After sending the File Assigner requests the Task Scheduler returns control to the Run-Time Monitor without being preempted. The advantage of this integration model is that decision to send request to change the distribution of public file copies to the File Assigner can occur very early, through a Task Scheduler decision to not schedule a number of task instances. However, the basis for this decision is quite inaccurate since the Task Scheduler has only vague information about the availability of the needed resources. The interdependencies of the Task Scheduler-Oriented Integration model can be seen in figure 6-6.

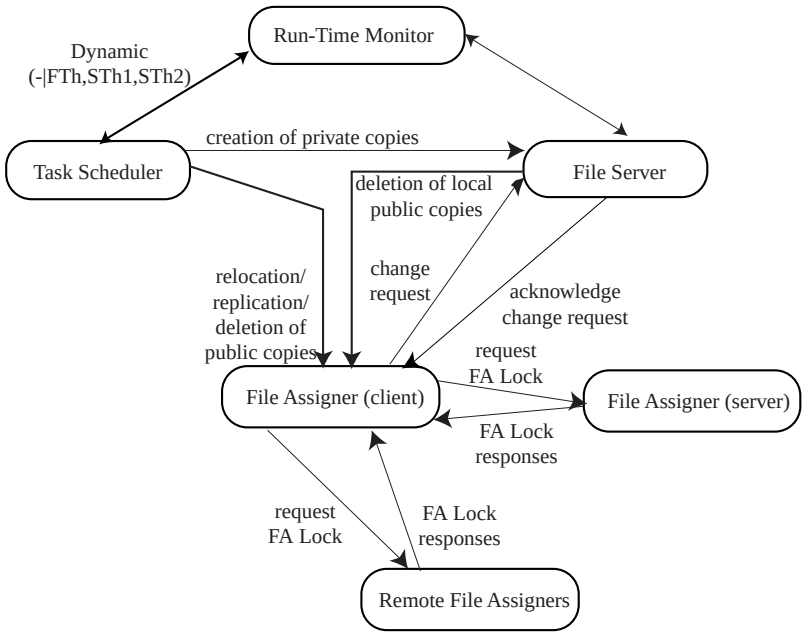


Figure 6-6: File Assigner: Task Scheduler-Oriented Integration Model

File Server-Oriented Integration. In this model the File Server (see section 4.1.5) maintains three queues of task instances that missed their deadline: for *sensitive read tasks* (**FSR**), for *write tasks* (**FSW**), and for *robust read tasks* (**FRR**). Each of the failed task instances is kept in the corresponding queue for a fixed amount of time. Thus the contents of a queue represent a set of task instances that have (actually) failed during a fixed time window. Upon completion of a updating the file history, the File Server sends requests to the File Assigner based on the following threshold values:

FSWTh: When the number of task instances queued in FSW surpasses this threshold the File Server requests to reduce the number of public copies for this specific file. This would have the effect of improving the performance of task accessing this file by reducing the time required to access the public copies of this file.

FSRTh: Once the number of task instances queued in FSR surpasses this threshold the File Server requests to create a local public copy for this specific file that currently is not available at the local site. This improves the performance of reading task instances at the site (since local access time is much faster then remote access with its additional communication overhead).

FRRTh: When this threshold has been surpassed by the number of task instances queued in FRR the File Server requests to create a local private copy for the specific file. The performance of these task instances would then be improved for the same reason regarding local access times stated for FSRTh.

As in the Task Scheduler-Oriented model the File Server maintain control over determine when

to delete local public and private copies based on the *delete copy threshold*. Creation of private copies and deletion of public copies under emergency conditions would also be handle in the same method whenever it was detect that the next instance of a task would become essentially critical. Thus, the Task Scheduler has no further connection with File Assigner, and the File Server acquires a very similar function without changing its other structures. While decision to request the File Assigner to change the distribution of public file copies occurs at a later point of time than in the Task Scheduler-Oriented model the basis this decision is more accurate since the information regarding the files is more accurate. The interdependencies of the File Server-Oriented Integration model can be seen in figure 6-7.

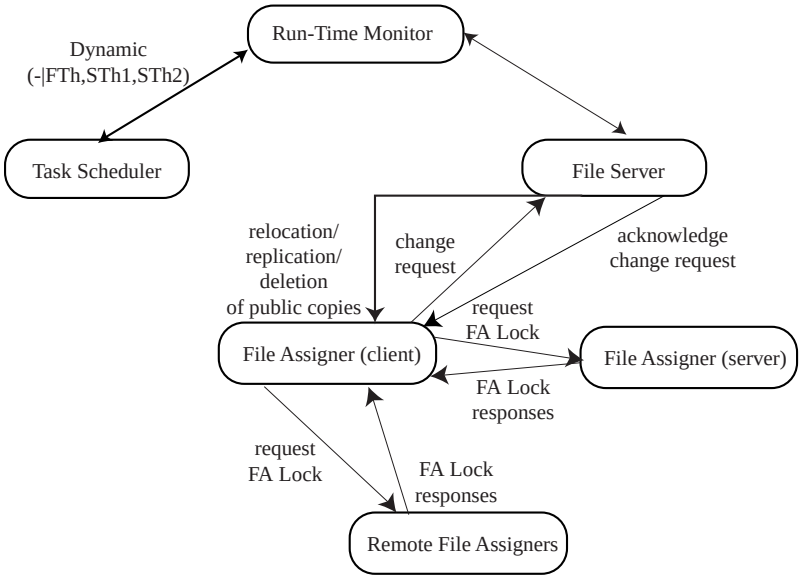


Figure 6-7: File Assigner: File Server Oriented Integration Model

The actions taken by either the File Assigner client or server are very short in duration, while for the remaining time both the File Assigner client or server are waiting for remote requests or responses. Therefore, the invocation of the File Assigner (client and server) is controlled by the File Server, interleaving File Server and File Assigner requests. All requests are separated into two distinct categories, *essentially critical requests* (on behalf of essentially critical task), and *non-essential requests*. Both queues are further separated into *current access requests* and *future assignment requests*. Current access requests contain all requests from or to task instances or files located at the site. Future assignment requests contain all local or remote File Assigner requests and responses for the relocation, replication and deletion of shared files. The File Server processes requests until it has to stop under the rules of File Server/Task Scheduler integration (see section 7.1.1). The interleaving of File Server access requests and File Assigner actions are prioritized in the following order:

1: The File Server handles all current access requests in the essentially critical request queue.
2: The File Assigner handles all future assignment requests in the essentially critical request queue.
3: The File Server handles all current access requests in the non-essential request queue.
4: The File Assigner handles all future assignment requests in the non-essential request queue.

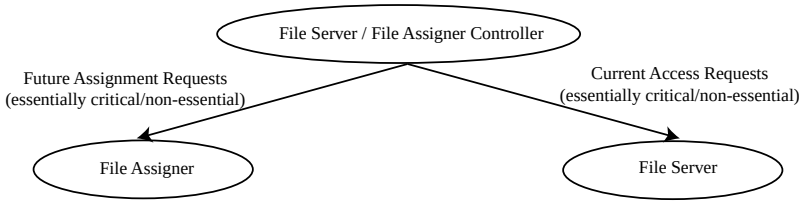


Figure 6-8: File Server/File Assigner Control Integration

6.2 File Assigner Implementation

As a summary of the presentation in the previous section the File Assigner is partially presented in pseudo code.

6.2.1 File Assigner Client Implementation

The following protocol services assist the local FA client in completing the specific protocols used to change the distribution in response to the request made by the decision services. This includes the services: *Check Move Copy, Check Create Copy, Check Delete Copy.* Also included are the services required to request or release the FA Lock: *Request FA Lock, Granted FA Lock, Release FA Lock, Released FA Lock.*

Check Move Copy. Using this service the FA client checks to see if the FA servers have allowed for a public copy to be moved from a remote site to the local site. The algorithm 6-4 first set the status of the move request to true (meaning the movement is granted). It will then go through all locations that have a public copy located at their site. If any location has denied the local site the ability to move its public copy to the local site then the status is set to false. At the end, the value of the move status is returned. A value of true means that the FA client is allowed to begin moving the public copy at a granting site to the local site. A false value means that at least one remote FA server has denied the local FA client's request to move a public copy to the local site.

```
Check Move Copy(File  F)
{
   Status = TRUE;
   for all locations of F {
      if LOCATION_DENIED_MOVE(location) {
         Status = FALSE; } }
   return(Status);
}
```

Algorithm 6-4: Check Move Copy

Check Create Copy. The FA client checks if the FA servers that have a public copy have allowed for a public copy to be created at the local site. The algorithm 6-5 first set the status of the creation request to true. The algorithm then goes through all locations that have a public copy located at their site. If any location has denied the creation of a new public copy, the status is set to false. At the end of the algorithm the value of the status is returned. A true value means that the FA client is allowed to begin to create a new public copy at the local site. A false value means that the local FA client's request to create a public copy is denied.

```
Check Create Copy(File  F)
{
   Status = TRUE;
   for all locations of F {
      if LOCATION_DENIED_COPY(location) {
         Status = FALSE; } }
   return(Status);
}
```

Algorithm 6-5: Check Create Copy

Check Delete Copy. Using this service the FA client checks to see if the FA servers have allowed for a public copy to be deleted from a site that has a public copy. First the status of the reduction request is set to true by the algorithm 6-6. It then goes through all locations that have a public copy. If any location has denied that its local public copy could be deleted the status is set to false. At the end of the algorithm the value of the status is returned. A true value means that the FA client is allowed to begin to delete a public copy. A false value means that a remote FA server has denied the local FA client's request to reduce the number of public copies.

```
Check Delete Copy(File  F)
{
   Status = TRUE;
   for all locations of F {
      if LOCATION_DENIED_DELETE(location) {
         Status = FALSE; } }
   return(Status);
}
```

Algorithm 6-6: Check Delete Copy

Request FA Lock. The FA clients use this service to send requests to all FA servers to obtain the

FA Lock on the public copy. First algorithm 6-7 marks the file that it is trying to get the FA Lock. It then sends a broadcast request to all FA servers to try and obtain the FA Lock. After this message is sent, the FA client waits for the FA servers (local and remote) to respond by either granting or denying the FA Lock from their sites.

```
Request File Assigner Lock(File  F)
{
   mark_file_locking(F);
   request_FA_lock_file(F);
}
```

Algorithm 6-7: Get FA Lock

Granted FA Lock. When the FA client receives a response from an FA server this service is invoked to handle the granted FA Lock. The algorithm 6-8, first marks it has received a FA Lock from a site holding a public copy. The algorithm then checks to see if all locks have been returned. If all locations have been locked, then the algorithm returns true (meaning that the FA Lock has been granted to the local FA client). Otherwise, the algorithm returns a value of false (meaning that the FA Lock has not yet been granted to the local FA client).

```
Granted FA Lock(File  F, Host I)
{
   mark_copy_locked(F,I);
   if (HAS_FA_CLIENT_LOCKED_ALL_COPIES(F) {
      return(TRUE); }
   else {
      return(FALSE); }
}
```

Algorithm 6-8: Granted FA Lock

Release FA Lock. Sends a request to release the FA Lock that the local FA client holds to all FA servers that have public copy. First algorithm 6-9 marks the request that it is releasing the FA Lock. It then sends a broadcast to all FA servers to release the FA Lock for the local FA client. After this is sent, the FA client waits for the FA servers (local/remote) to respond by either granting or denying the FA Lock from their sites.

```
Release File Assigner Lock(File  F)
{
   mark_file_releasing(F);
   request_release_FA_lock(F);
}
```

Algorithm 6-9: Release FA Lock

Released FA Lock. This service is invoked by the local FA client when it receives a response from a FA server that the FA server has released the local FA client's lock on its copy. The algorithm 6-10, marks that it has received a response stating that the FA Lock has been released at a specific site. The algorithm then checks to see if all locks have been released. If all locations holding a public copy have released the lock, then the algorithm returns true, otherwise it returns false.

```
Released FA Lock(File  F, Host I)
{
   mark_copy_released(F,I);
   if (HAS_FA_CLIENT_RELEASED_ALL_COPIES(F) {
      return(TRUE); }
   else {
      return(FALSE); }
}
```

Algorithm 6-10: Released FA Lock

6.2.2 File Assigner Server Implementation

The FA server has been implemented by separating the services provided into two categories: FA Lock handling and history decisions. **FA Locking** services handle all requests to lock or release the FA Lock on a local public copy. This includes the services: *Request FA Lock* and *Release FA Lock*. **History Decision** services provide the FA server the ability to decide whether to grant or deny a specific request to change the distribution of public copies. This includes the services: *Allow Replication*, *Allow Relocation* and *Allow Deletion*.

Allow Replication. This service decides for the FA server whether it allows a FA client to create a new public copy at another site. In algorithm 6-11, the total number of failing write access requests is compared to the threshold *Allow_Copy_Threshold*. If the value is underpassed, then the algorithm returns the value true (stating that the FA server allows the remote FA client to create a public copy). Otherwise, the algorithm returns a value of false.

```
Allow Replication(File F)
{
   if (W.Failures < Allow_Copy_Threshold) {
      return(TRUE); }
   else {
      return(FALSE); }
}
```

Algorithm 6-11: Allow Replication

Allow Relocation. To decide whether an FA client is allowed to move the local public copy to another site the FA server calls this service. In algorithm 6-12, the total number of access requests by sensitive and essentially sensitive reading task instance is compared to the threshold Delete_Copy_ Threshold. If the value is underpassed, the algorithm then returns the value true (stating that the FA server allows for the relocation of its local public copy). Otherwise, the algorithm returns a value of false.

```
Allow Relocation(File F)
{
   if ((RS.Failures + RS.Success) < Delete_Copy_Threshold) {
      return(TRUE); }
   else {
      return(FALSE); }
}
```

Algorithm 6-12: Allow Relocation

Allow Deletion. The following service decides for the FA server whether to allow the FA client to delete the public copy located at its site. In algorithm 6-13, the total number of access requests by reading task instance is compared to the threshold *Delete_Copy_Threshold*. If the value is underpassed, the algorithm then checks if there are more than the minimum number of public copies allowed. If there are more than the minimum, then the algorithm returns the value true (stating that the FA server has allowed for the remote FA client to delete the public copy located at its site). Otherwise, the algorithm returns a value of false.

```
Allow Delete(File F)
{
   if ((((RS.Failures + RS.Success) + (RR.Failures + RR.Success)) < Delete_Copy_Threshold)
        AND (Copies(F) > Minimum_Public_Copies)) {
      return(TRUE); }
   else {
      return(FALSE); }
}
```

Algorithm 6-13: Allow Delete

Request FA Lock. This service is invoked by the local FA server when it receives a request from a FA client to obtain the FA Lock on the public copy located at the local site. In algorithm 6-14, a request to obtain to the FA Lock for a specific file is received by the FA server. If the FA Lock is free then the lock request is granted to the requesting site. This granted FA Lock includes the file history information. If the FA Lock is not free, the algorithm then checks to see if the requestor has a higher priority then the lock holder. If the requestor does have a higher priority, then the old lock holder is denied the lock by sending a denied lock response. The requesting site is then placed into a waiting state and waits until the lock is released. In all other cases the requestor is denied the FA Lock.

```
Request FA Lock(File  F, Host I)
{
   if IS_FA_LOCK_FREE(F) {
      grant_FA_Lock(I); }
   else {
      if IS_REQUESTOR_GREATER_PRIORITY(I) {
         callback_FA_Lock(OLD_FA_LOCK_HOLDER);
         wait_FA_Lock(I); }
      else {
         wait FA Lock(I); } }
}
```

Algorithm 6-14: Request FA Lock

Release FA Lock. Once invoked by the local FA server when it receives a request from a FA client to release the FA Lock on the public copy that the FA client holds, the algorithm 6-15 releases the FA Lock for the specific file. If another FA client is waiting to obtain the FA Lock, then that FA client is granted the lock, otherwise the lock is released.

```
Release FA Lock(File  F, Host I)
{
   remove from FA Lock queue(I);
   if IS_FA_LOCK_WAITING(F) {
      grant_FA_Lock(WAITING_FA_LOCK_HOLDER); }
   else {
      free_FA_Lock(F); }
}
```

Algorithm 6-15: Release FA Lock

Chapter 7 Run-Time Monitor

In traditional operating system design, resource acquisition is done prior to task scheduling. A task instance that had achieved locks on all its required resources would then be scheduled/guaranteed by the Task Scheduler, or otherwise be subject to abortion. Throughout this entire time competing task instances would be blocked and may run out of time. This remote blocking would be *uncontrollable* by their local agencies. Distributed resource allocation occupies a comparably large portion of the task execution time, in contrast to the purely local task scheduling activities. Thus the abortion decision by the Task Scheduler would come very late. To instead abort tasks as early as possible and at the same time lock the resources as late as possible the principle was established to *reverse the order of task and resource scheduling* in MELODY (see section 1.2). This leaves the Task Scheduler without accurate information on task execution times (the actual resource allocation time is unknown when the Task Scheduler is invoked) and a task instance would then have to be scheduled based on estimates. As a result the Task Scheduler can no longer guarantee a task instance will meet its deadline (since resource scheduling would be completed after task scheduling). However, by introducing a novel **Run-Time Monitor** (RTM) module not only would the Task Scheduler abort task instances, but the Run-Time Monitor would supervise resource acquisition and would abort task instances as early as possible during their acquisition phase. This abortion would be an accurate decision since the execution time is known before the locking procedure begins. At the same time competing task instances would benefit from resources being locked as late as possible (after the task scheduling phase). Experiments conducted using the simulator developed in phase 5 (see section 1.2) showed significant benefits (both in terms of deadline performance and survivability) in this reversed task and resource scheduling policy when compared to the *classical* model (task scheduling prior to resource scheduling) [WeK93].

To satisfy the tight constraints found in safety-critical real-time systems it is necessary to control all activities that could effect a task instance's ability to meet its deadline. Since the purely local task scheduling activity takes much less time than (remote) file acquisition, integration of task and resource scheduling means most likely to invoke the Task Scheduler during the resource scheduling activities of the File Server. Integration of task scheduling activities would have to handle the following conflicting goals:

- Infrequent invocation of the Task Scheduler. This minimizes the overhead caused by Task Scheduler activities (increasing the time for resource scheduling activities) and the context switching overhead while at the same time the number of task instances waiting to be scheduled is increased thus providing for 'wiser' scheduling decisions (selection from a larger set of waiting task instances). This policy equally means to invoke the Task Scheduler *as late as possible*.
- Invocation of the Task Scheduler *as early as possible*. This increases, for every individual task instance, the time available for acquiring the needed resources.

To adaptively handle these issues a dynamic integration policy under the control of the Run-Time Monitor was developed (see section 1.2) which would invoke the Task Scheduler based upon the level of competition and the number of task instances waiting to be scheduled. A technically detailed presentation of the Run-Time Monitor activities will be given in the following sub-sections.

7.1 Run-Time Monitor Model

The distributed Run-Time Monitor has been designed [Seg97] to monitor three main features of the MELODY system: Tasks, Files and Integration Policies. To accomplish these tasks the Run-Time Monitor modules have been separated into three sub-servers (see figure 7-1):

Integration Controller: This sub-server has been designed [WeS96, WeL97 and WeL98] to control the dynamic Task Scheduler and File Server integration policy in MELODY. The integration

controller determines at which points of time the Task Scheduler should be invoked to schedule a set of newly arrived task instances. Detailed descriptions of the policies used in making this decision can be found in section 7.1.1. A detailed experimental evaluation of the policies used for Task Scheduler and File Server integration can be found in chapter 9. The integration controller also inspects the Task Scheduler's queue of newly arrived task instances (Task Scheduler *waiting queue* (***TSW***)). It removes any task instance that is essentially critical from this queue, and immediately places it into the File Server's *competing task queue* (***CTQ***) (these task instances have priority over all other tasks).

Task Monitor: This sub-server objective is to abort a task instance T_{jk}, that no longer has a chance to complete before it's deadline, as soon as possible. This abortion of T_{jk} is based on estimates of the time required to complete resource location, allocation and locking for T_{jk}. The policies used are detailed in section 7.1.2. Their experimental evaluation can be found in chapter 11.

File Monitor: This sub-server monitors requests from File Server (local/remote) for access to files located at the site. If the File Monitor determines that the request can no longer be satisfied by this site (deadline of the request has expired), then the request is removed. File monitoring is necessary since the File Servers do not ensure that a *release* request sent to the location of a file is received and acknowledged. A detailed description of this function can be found in section 7.1.3.

Each sub-server is then directly responsible for one of the main features provided by the Run-Time Monitor.

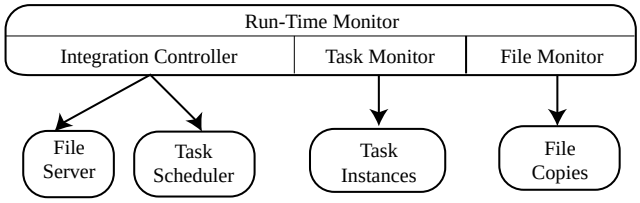


Figure 7-1: Run-Time Monitor Sub-Servers

7.1.1 Task Scheduler/File Server Integration Controller Sub-Server

It is expected that if the Task Scheduler is invoked too frequently that added invocations would result in degraded performance. This degraded performance results from two major influences: increased overhead required to invoke and complete Task Scheduler operations, and increased number of task instances competing for shared resources (due to less reliable information used during task scheduling). Delaying the invocation of the Task Scheduler for a longer period would improve the Task Scheduler's scheduling process. However, this increased delay in a task instance's task scheduling phase (see section 3.3) could also have significant effects on the chances of a task instance to complete prior to its associated deadline, by reducing the time allowed to complete the remainder task life cycle phases. We developed the following integration models:

Periodic model: In this model the TS is invoked after a preset interval of time has passed.

Dynamic model: The Task Scheduler function would not be of much use for scheduling essentially critical task instances since abortion of such tasks potentially causes the whole system to decease. Consequently, in MELODY the RTM also, while inspecting the TS queue of arrived task instances, picks the essentially critical instances and inserts them (in appropriate scheduling order) in to a new FS queue, the *essentially critical request queue* (ECR) (So the essentially critical tasks would bypass the Task Scheduler). The task instances that are still scheduled by TS (which are *critical* or *non-critical*) are inserted into the *non-essential request queue* (NER) at FS, *after a TS invocation took place*. While ECR and NER together form the set of tasks on which FS operates, ECR has priority over NER. Three dynamic thresholds are used to determine when to invoke the TS:

- A threshold **FTh** is set for the File Server's *Competing Task Queue* (*CTQ*).
- Two thresholds **STh1** and **STh2** (**STh1** < **STh2**) are set for the Task Scheduler's *Waiting Queue* (*TSW*). These two thresholds monitor the number of newly arrived tasks instances.

The Dynamic model then invokes the Task Scheduler whenever either of the following

conditions becomes true:

- **FTh** has been underpassed and **STh1** has been surpassed,
- **STh2** alone has been surpassed.

The threshold values are conceived to be adaptable, over certain time intervals, to specifics of the task profiles encountered.

Adjusted model: Threshold values for the Dynamic model can result in Task Scheduler being invoked too frequently. Therefore, it may be beneficial to delay invocation for an interval of time as defined in the Periodic model. This model adjusts the Dynamic model by only invoking the TS at certain predefined instances in time. The thresholds **FTh**, **STh1** and **STh2** (as defined in the Dynamic model) are used to determine when to invoke the Task Scheduler. However, this model delays invocation until the next period (defined in the Periodic model).

The main expectation for the new integrated task and resource scheduling policies is that the lack of guarantee by TS would be more than made up by the less chaotic and harmful way of early resource locking (as used in traditional operating system design). Our first concern was to compare any of the new integrated models with a classical model of periodic TS invocation and resource scheduling prior to task scheduling. This was subject to extensive simulation experiments reported in [WeL94].

It was expected that the Periodic model would have an optimal setting of the invocation period with respect to the deadline failure rate. This optimal performance would degrade as the period was increased or decreased from the optimum. For the Dynamic model there should also be optimal settings for FTh, STh1 and STh2. The Adjusted model had been developed to assure that under certain extreme settings its performance would match one of its constituent models. For example, a very long period for the Adjusted model would cause it to be significantly influenced by the period, and as a result its performance should be nearly identical to the Periodic model. The Adjusted model under a very short period should match the performance of the Dynamic model. In this way the characteristic tendencies of the Adjusted model could be directly attributed to the either the Dynamic model or the Periodic model, and this hybrid model could then be utilized as a benchmark model for comparatively evaluating the other two models.

7.1.2 Task Monitor Sub-Server

The Task Monitor sub-server of the Run-Time Monitor is responsible for monitoring task instances that are competing to access shared files. This monitoring is done by determining an estimate of the remaining time required to acquire all needed resources and complete the computation phase for the task instance. If the Task Monitor determines that the task instance can no longer complete its computation phase prior to its deadline then it aborts the task instance. The estimate used to determine the remaining time required to acquire all needed resources is based on the history of the prior task instances. The Task Monitor uses the following three estimates to determine the remaining time required by a task instance T_{jk}:

Estimated Acquisition Time (EAT_{jk}): This is the estimated time required by T_{jk} in order to acquire the locks for all needed resources.

Estimate Locking Time (ELT_{jk}): For a writing task instance T_{jk} the Task Monitor also determines an estimate for the time required to obtain the lock once all ready messages have been received by the task instance (once phase 1 of the Delayed Insertion protocol (see section 4.1.1) has been completed).

Estimated Computation Time (ECT_{jk}): This is the estimated time required to complete the operation requested by the task. For reading task instances this is the time at which the message is received at the local site. For writing task instances this is the time at which the update has completed on the remote site.

Once estimates for T_{jk} have been determined, the RTM sets a number of sub-deadlines that correspond to stages during the task execution life cycle (see section 3.3). This estimate is based on the deadline DT_{jk}, which is determined by the application. The following sub-deadlines then allow the Task Monitor to check which phase the task instance is trying to complete and whether the corresponding sub-

deadline has expired:

Location Sub-Deadline (DLo_{jk}): This sub-deadline is set based on task instance T_{jk}'s ECT_{jk} and EAT_{jk}. The deadline DLO_{jk} is determined by subtracting ECT_{jk} and EAT_{jk} from the value of DT_{jk}.

Acquisition Sub-Deadline (DAc_{jk}): This sub-deadline is set based only on the task instance T_{jk}'s ECT_{jk}. The deadline DAC_{jk} is determined by subtracting ECT_{jk} from the value of DT_{jk}.

Allocation Sub-Deadline (DAl_{jk}): This sub-deadline is set based only on the task instance T_{jk}'s ECT_{jk} and ELT_{jk}. The deadline DAl_{jk} is determined by subtracting ECT_{jk} and ELT_{jk} from the value of DT_{jk}. This sub-deadline is only set for writing task instances since a reading task instance is not required to obtain ready messages from the files in its List of Required Files (LRF_{jk}).

A graphical representation of the points at which a sub-deadline would be set for a reading task instance can bee seen in figure 7-2. The sub-deadlines for a writing task instance can be seen in figure 7-3 (note that read task instances have to acquire a read lock on one of the file copies (see section 3.3)). In both figures, CT_{jk} is the creation time of task instance T_{jk}. When the Task Monitor has determined that a task instance T_{jk} can no longer meet one of its associated sub-deadlines (and therefore its associated deadline DT_{jk}) then the Task Monitor aborts the task immediately.

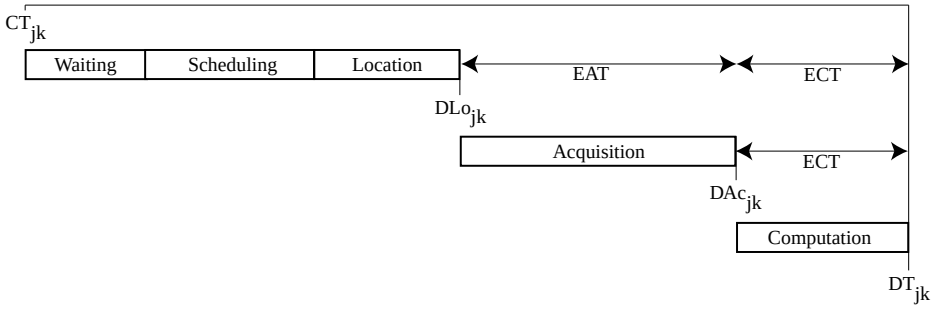


Figure 7-2: Reading Task Instance Sub-Deadlines

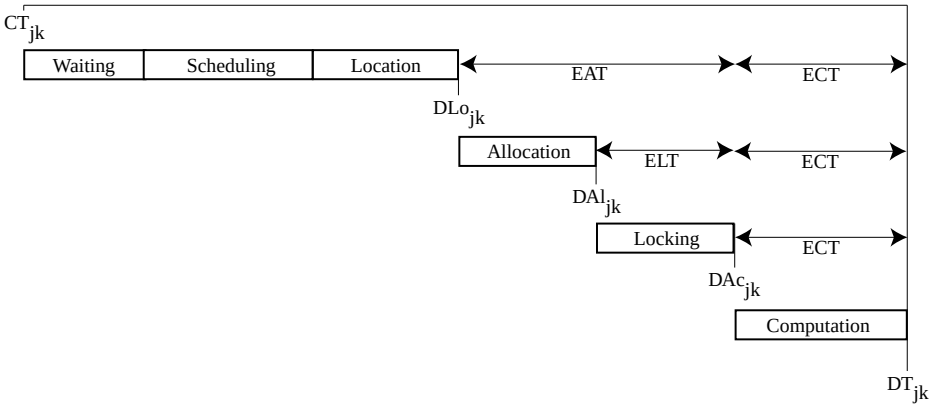


Figure 7-3: Writing Task Instance Sub-Deadlines

7.1.3 File Monitor Sub-Server

The File Monitor sub-server of the Run-Time Monitor is responsible for monitoring copies of files located at the local site. The monitor determines if a queued request (both for read/write access) from a task instance (local/remote) would still be able to obtain the required lock (in time) to complete before its associated deadline. If it determined that the requesting task instance no longer has a chance to obtain the required lock and complete its computation phase then the queued request will be removed from the file manager's queues according to the *Delayed Insertion protocol* (see section 4.1.1). Monitoring is done to improve the response times of the local file copy, since there would be less blocking requests by a request which has no chance to complete before its deadline. A removal of a task instance request is possible since the File Monitor only removes those request that have no chance to complete, therefore it would not be necessary for the local File Server to wait for the *release* message from the task instance (that may be delayed by the communication load which would add additional time to the already unnecessary blocking time). Any *release* message received after removal of the request would then be ignored by the local File Server.

7.1.4 Run-Time Monitor Integration

The actions taken by the sub-servers of the Run-Time Monitor are very short in duration, since they are oriented towards checking whether a specific action should be taken and therefore these services are non-preemptive. Integration of these modules has then been directly integrated into the services of the File Server. This being done because it has the information on when the specific modules should be called to perform their checks. The File Server would then processes requests until it has to stop under the rules of File Server/Task Scheduler integration (see section 7.1.1). Invocation of the File Monitor or Task Monitor could be delayed by the invocation of the Task Scheduler (which may be up for invocation at the same time as the two services). If invocation of the Task Scheduler occurs before either service is invoked, then once the Task Scheduler completes its operations control would be returned to the File Server which would then invoke the delayed service (resuming at the point where it had been interrupted by the Task Scheduler's invocation). Since the invocation of all other MELODY modules is control by the File Server no other module could then interrupt the their invocation. During the File Servers processing it would invoke either the Task Monitor or File Monitor depending on the type of request being handled. The Task Monitor sub-server is called directly by the File Server to check the ability of a task instance to complete prior to its associated deadline and is directly integrated into the access routines of the FS client (see section 4.1.2) that control a task instance's movement through its task life cycle (see section 4.2.1). The Task Monitor is invoked for an individual task instance after the completion of a phase. This invocation checks the current phase the task instance has just completed and the sub-deadline (see section 7.1.2) that was assigned to that phase. Following this control is returned to the File Server such that it may be allowed to either continue handling the task instance or abort it. Invocation of File Monitoring services is then integrated into the file allocation routines of the FS server (see section 4.1.3). Invocation occurs whenever an access (read/write), schedule (write) or execution (read/write) request is received. When received, the File Monitor removes any outstanding requests to ensure that the proper response can be issued by the File Server to the received request.

7.2 Run-Time Monitor Implementation

Remember that the implementation of the Run-Time Monitor module is separated into the three sub-servers: integration controller, Task Monitor and File Monitor. The integration controller is responsible for the following activities:

- Placing newly created task instances into either the Task Scheduler's *Waiting Task Queue* (*WTQ*), or placing essentially critical task instances into the File Server's *Competing Task Queue* (*CTQ*).
- Determining whether the Task Scheduler should be invoked based on the integration model.

The Task Monitor sub-server is responsible for setting sub-deadlines for all task instances in the File Server's *CTQ*, and for aborting those task instances as soon as possible based upon their corresponding sub-deadlines. The File Monitor monitors requests (from both reading and writing task instances) queued for access in the file manager's access queues (see the *Delayed Insertion protocol* in section 4.1.1), and it aborts any request that no longer has a chance to complete before its associated deadline. The relationships between the algorithms of the Run-Time Monitor services and the Task Scheduler and File Server services is graphically shown in figure 7-4.

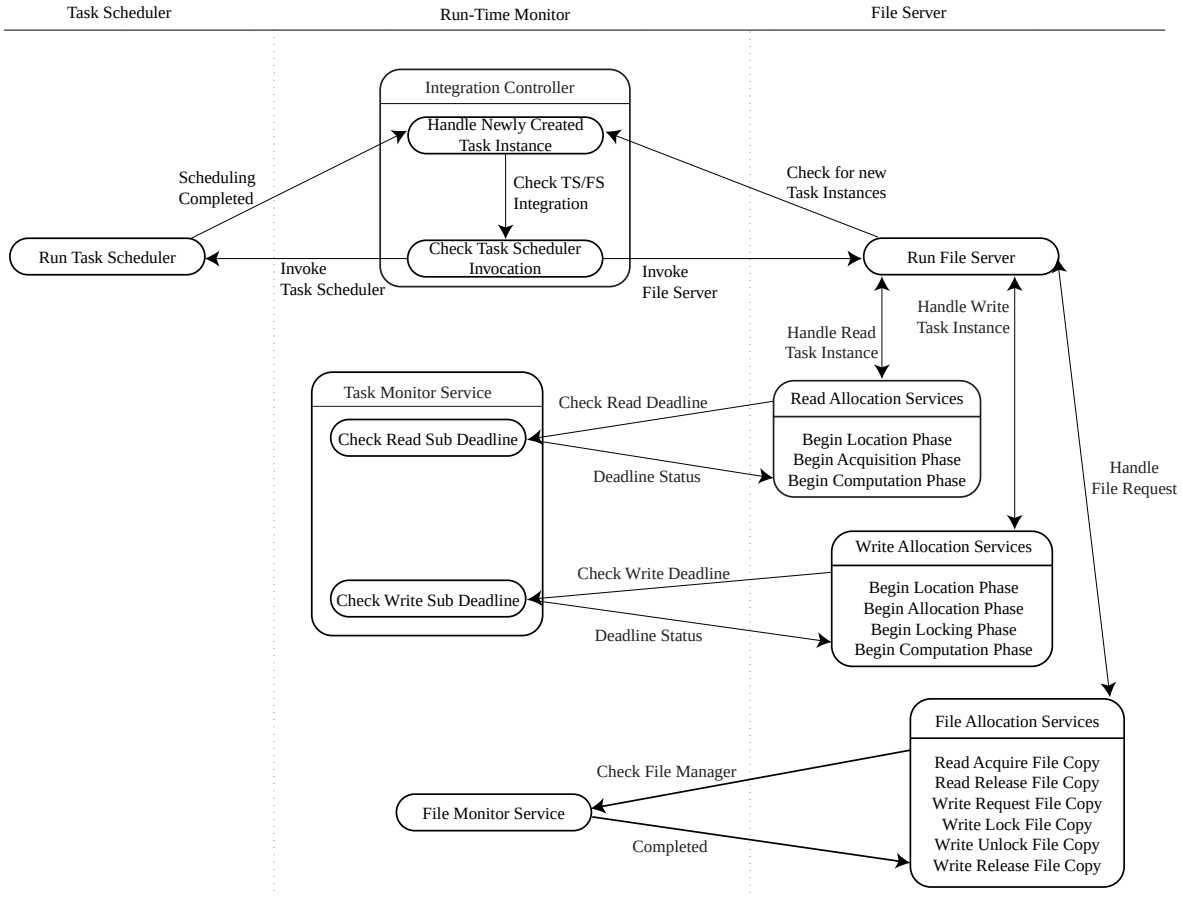


Figure 7-4: Run-Time Monitor Services Intergration

7.2.1 Task Scheduler/File Server Integration Controller Services

The integration controller is invoked by the Run-Time Monitor in order to check for any newly created task instances. A newly created task instance T_{jk} is assigned a relative degree of criticality and sensitivity based on the status of the prior task instance $T_{j(k-1)}$. If T_{jk} is determined to be essentially critical, then the integration controller queues the task instance into the File Server's *CTQ*. Otherwise, the task instance T_{jk} is placed into the Task Scheduler's *WTQ*.

The integration controller then checks to determine if the Task Scheduler should be invoked based on the model for TS/FS integration being used. If the Task Scheduler is not to be invoked, the Run-Time Monitor gives control to the File Server. Otherwise, the Integration controller invokes the Task Scheduler. Once the Task Scheduler has completed scheduling all task in its waiting task queue it gives control to the File Server. In the sequel we describe the relevant functions of the integration controller in pseudo-code.

Handle Newly Created Task Instance. This service checks for any task instance T_{jk} newly created by the application. It assigns it a relative degree of criticality C_{jk} (see section 3.1.1), and relative degree of sensitivity R_{jk} (see section 3.1.2), based on the status of the prior task instance $T_{j(k-1)}$. Based on the relative degree of criticality this service then determines if the task is to be placed into the Task Scheduler's waiting task queue, or the File Server's competing task queue.

```
handle_new_instance_creation()
{
    for all new task instances Tjk do {
      Tjk->Criticality = assign_criticality(Tjk);
      Tjk->Sensitivity = assign_sensitivity(Tjk);
      if (Tjk->Criticality is essentially critical) {
         Begin Location Phase(Tjk); }
      else {
         Begin Scheduling Phase(Tjk); } }
}
```

Algorithm 7-1: Handle New Instance Creation

Check Task Scheduler Invocation. This service checks to determine if the integration controller should pass control to the Task Scheduler in order for it to schedule all task instances in its waiting task queue, or give control to the File Server. The determination of whether to invoke the Task Scheduler or not is based upon the definition of the integration model selected: Periodic, Dynamic or Adjusted (see section 7.1.1).

If the Periodic model is being used, the algorithm then checks to determine if the interval has elapsed. If the interval has elapsed, then the Task Scheduler should be invoked and a successful status is returned, otherwise the File Server should be given control and a failed status is return. If the Dynamic model is being used then the algorithm checks if the threshold STh2 has been surpassed, or if FTh has been underpassed and STh1 has been surpassed. If the Task Scheduler should be invoked by the Dynamic model then a successful status is returned, otherwise a failed status is return. If the Adjusted model is being used, the algorithm first checks if the interval has elapsed or not. If the interval has elapsed and the Dynamic model would have invoked the Task Scheduler then a successful status is return, otherwise a failed status is returned. In all cases if a successful status is returned by this algorithm the Integration Controller immediately gives control to the Task Scheduler, otherwise control is given to the File Server.

```
Status_T    Check Task Scheduler Invocation()
{
   switch (TASK_SCHEDULER_CONFIG->Method) {
   case PERIODIC: if CHECK_PERIODIC(TASK_SCHEDULER_CONFIG->Periodic) {
                     UPDATE_PERIOD(TASK_SCHEDULER_CONFIG->Periodic);
                     return(SUCCESS); }
                  else {
                     return(FAILED); }
                  break;
   case DYNAMIC : if CHECK_DYNAMIC(TASK_SCHEDULER_CONFIG->Dynamic) {
                     return(SUCCESS); }
                  else {
                     return(FAILED); }
                  break;
   case ADJUSTED: if CHECK_PERIODIC(TASK_SCHEDULER_CONFIG->Periodic) {
                     UPDATE_PERIOD(TASK_SCHEDULER_CONFIG->Periodic);
                     if CHECK_DYNAMIC(TASK_SCHEDULER_CONFIG->Dynamic) {
                        return(SUCCESS); }
                     else {
                        return(FAILED); } }
                  else {
                     return(FAILED); }
                  break; }
}
```

Algorithm 7-2: Check Task Scheduler Invocation

7.2.2 Task Monitor Services

The Task Monitor sub-server is invoked directly by the File Server in order to check the ability of a task instance to complete its phase prior to its associated deadline. Invocation is highly integrated into the access routines to control a task instance's movement through its task life cycle (see section 4.2.1), therefore overhead required to invoke task monitoring services is at minimum and is considered negligible. The Task Monitor is invoked for a task instance after the completion of each of its phases. This invocation checks the current phase the task instance has just completed and the sub-deadline (see

section 7.1.2) that was assigned to that phase. Following the monitoring of the task instance control is returned to the File Server such that it may be allowed to either continue handling the task instance or abort the task instance. Since the time required by task monitoring is very short in duration this service is non-preemptive. However, invocation of the Task Monitor is preemptive and could be delayed by the invocation of the Task Scheduler (which may be up for invocation at the same time as the Task Monitor). If invocation of the Task Scheduler occurs before the Task Monitor is invoked, then once the Task Scheduler has completed its operations control would be returned to the File Server which then invoke the Task Monitor service (resuming at the point where it had been interrupted by the Task Schedulers invocation). Since the invocation of all other MELODY modules is control by the File Server no other module could then interrupt the invocation of the Task Monitor.

If the sub-deadline can not be met, then the Task Monitor assumes that the task deadline can also not be met, and aborts the task instance. The sub-deadlines checked by the Task Monitor are based on the estimates for the time required by the task instance to complete a certain phase in its task life cycle (see section 7.1.2): estimated acquisition time (EAT_{jk}), estimated locking time (ELT_{jk}) and estimated computation time (ECT_{jk}). In MELODY all of the estimates are determined by utilizing the task history for a given task T_j. Depending on the type of estimate used the Task Monitor either utilizes a minimum, average or maximum value for the phase being estimated. The type of value utilized is envisioned to be tailored to the application. However, in case of the estimated computation time only an average is utilized. This is due to the fact that the time to complete the computation phase for both writing and reading task instances is fairly constant (for a given distribution of the needed file copies). In case of a reading task instance the computation request always consists of one execution request being sent to the node which holds the selected file copy to perform the computation against, and a response received with the result of the computation. For writing task instances the computation phase consists of one broadcast message being sent to the nodes involved in the update operation. The time used by the writing task instance is then only until the operation has been completed at all nodes.

The Task Monitor utilizes two services: *check write sub-deadline* or *check read sub-deadline*, to check the deadline for whether a writing or reading task instance T_{jk}. Based on the return values from these two services, the Task Monitor either allows the task instance to continue, or it aborts it from the File Server's *Competing Task Queue* using the service *begin task abort* (see section 4.2.1.1).

Check Write Sub Deadline. This service checks whether a writing task instance still has a chance to complete the phase that it's in. If the phase can be completed, and the associated sub-deadline can still be met, then the service returns a SUCCESSFUL status, otherwise a FAILED status is returned and the instance is aborted.

```
Status_T    Check Write Sub Deadline(Instances Tjk)
{
   switch (Current_Phase(Tjk)) {
   case LOCATION :   Current_Sub_Deadline = Tjk->DLojk;
                     break;
   case ALLOCATION : Current_Sub_Deadline = Tjk->DAljk;
                     break;
   case LOCKING :    Current_Sub_Deadline = Tjk->DAcjk;
                     break; }

   if (CURRENT_TIME > Current_Sub_Deadline) {
      return(FAILED); }
   else {
      return(SUCCESS); }
}
```

Algorithm 7-3: Check Write Sub-Deadline

The phase of the task instance T_{jk} is checked by the algorithm to determine whether the task instance is in either the location, allocation or locking phase. If T_{jk} is in the location phase, the algorithm then checks if the CURRENT_TIME has surpassed the location sub-deadline ($\mathbf{DLo_{jk}}$). If surpassed, then T_{jk} is aborted, otherwise it is allowed to continue. If T_{jk} is in the allocation phase (phase 1 of the Delayed Insertion protocol), the algorithm then checks if the CURRENT_TIME has surpassed the allocation sub-

deadline ($\mathbf{DAl_{jk}}$). If sub-deadline $\mathbf{DAl_{jk}}$ has been surpassed, then T_{jk} is aborted, otherwise it is allowed to continue. If T_{jk} is in the locking phase (phase 2 of the Delayed Insertion protocol), the algorithm then checks if the CURRENT_TIME has surpassed the acquisition (end of locking phase) sub-deadline ($\mathbf{DAc_{jk}}$). If sub-deadline $\mathbf{DAc_{jk}}$ has been surpassed, then T_{jk} is aborted, otherwise it is allowed to continue. In all cases the algorithm returns the value of failed meaning that the task instance T_{jk} should be aborted, or success meaning that T_{jk} should be allowed to continue.

Check Read Sub Deadline. This service checks whether a reading task instance T_{jk} still has a chance to complete the phase that it is in. If the phase can be completed, and the associated sub-deadline can still be met, then the service returns a successful status, otherwise a failed status is return and the task instance is aborted.

```
Status_T    Check Read Sub Deadline(Instances Tjk)
{
   switch (Current_Phase(Tjk)) {
   case LOCATION :    Current_Sub_Deadline = Tjk->DLojk;
                      break;
   case ACQUISITION : Current_Sub_Deadline = Tjk->DAcjk;
                      break; }

   if (CURRENT_TIME > Current_Sub_Deadline) {
      return(FAILED); }
   else {
      return(SUCCESS); }
}
```

Algorithm 7-4: Check Read Sub Deadline

7.2.3 File Monitor Services

The File Monitor sub-server is invoked directly by the File Server in order to check the validity of requests (from local or remote task instances). If the request is no longer valid (the deadline on the request has elapsed) then the request is removed. The removal of the request is handled in the exact same method as if the task instance that requested the file had sent a *release request* message to the file manager. Requests are checked by the File Server whenever a change in the status of the file happens. Status changes for file copies occur whenever a reading task instance's request arrives at either a shadow or private copy. Status changes also occur when a write lock is released on a public copy, or there are no longer any task instances in the scheduling queue.

Chapter 8 File System Experiments

To test the performance of the distributed MELODY implementation a set of experiments was performed that replicated the previous simulation experiments conducted in phase 5 of MELODY's development [WeK93]. These experiments compared the functionality of MELODY's file system to simpler, yet less flexible, models that exhibit some but not all of MELODY's functionality. The experiments compared the MELODY file system to two simpler file system models, *Public* model and *Private* model. The Public model only allowed for the relocation, replication and deletion of public copies. No private copies are provided by the Public model. Every update to a file copy invoked a set of file servers managing the public copies under the strong concurrency requirements detailed in section 3.2. The Private model allowed for the creation and deletion of private copies, but did not allow for any relocation, replication or deletion of public copies (rather providing for only a minimum number of public file copies of each file). This gave rise to a concurrency protocol that required less communication overhead (no distributed updates, no consensus procedures). However, it was costly to *refresh* private copies (through transferring a copy from a remote site) if updates to a file occurred very frequently.

For the purpose of the MELODY simulations a number of simplifications had been made to the system model. The effect that communication, real task computation and file manipulation would have as compared to the simulation model, was an open question (section 1.2). In particular, since the distributed model had been implemented on a single processor it was not possible to realistically implement distributed task computation and file manipulation, or to study the real overhead caused by invoking the corresponding system services. Also, actual communication within a distributed real-time environment is designed to have a strict upper bound. As the load on the real communication medium increases (due to unpredictable task arrival rates, file update frequency or file transfers) so does the potential for the communication time to exceed any designed bounds. The load on the communication medium also effects the basic assumption that no message is lost and that the order of messages is preserved for synchronization messages.

The results obtained during these prior simulation experiments were well understood and exhibit distinct characteristic tendencies. All aspects considered it was hoped that by modifying and varying model parameters, task and data profiles, the simulation set-ups and assumptions (communication times, file manipulation and task computation) could be modeled as extreme circumstances in the distributed set-ups. At the same time it was expected that the results/tendencies of the simulation studies could be recovered under the corresponding extreme distributed set-ups and would be refinable or extendible to the general distributed experimental situation. In this way using incremental experimentation the previous well-understood results could be utilized to evaluate differences in the performance of MELODY under real communication, task computation and file manipulation in a distributed environment.

Previous experiments showed, for task profiles with high sensitivity values (close to the threshold of essentially sensitive), that the Public model performed better than the Private model (especially under a high dominance of read operations). However, it was shown that the influence a task sensitivity is so strong that it makes the roles of the Public model and the Private model flip in terms of their relative deadline failure rate performance. In turn, for ranges of decreasing sensitivity (toward robustness) the Private model is eventually able to improve its performance and surpass the Public model's deadline failure rate performance. The performance increase can be attributed to the low overhead of the Private model for managing public file copies, while there is little overhead being

generated by refreshing public copies. The situation changes significantly as the number of write operations increases. The Private model no longer outperforms the Public model under decreasing levels of sensitivity. Also under a high dominance of write operations both models fail completely (survivability) even for ranges of moderate criticality. Throughout all the simulation runs, MELODY's performance was distinctively superior over the simpler models. It failed only when there was a high dominance of write operations and a high level of criticality in the task profile.

Experiments conducted using the simulator were redone by setting up a task profile that, by modify task profile parameters, would model (under extreme settings) the task profile used by the simulation experiments. In the experiments discussed here, 7 sites were used with 100 task distributed among all sites. 16 data files were distributed among all sites, each file initially with one public copy and no private copies. Parameter ranges for the task profiles (from which initial settings for the tasks in the profile were made) are shown in table 8-1. In the following task profiles, task deadlines had then been modeled accordingly to fit tightly. The rather high deadline failure rates (which can be seen in the following figures) were thus to be expected. They are not unacceptable as long as survivability is guaranteed, i.e. the failed the failed task instances had not yet been essentially critical. In this way characteristic differences in the performance of the models under a relatively high load could be more accurately determined. The threshold values for criticality a_i' and a_i" were uniformly set to 1 and 10, respectively, across all sites. Across all sites, the threshold values for sensitivity b_i' and b_i" were uniformly set to 5 and 10, respectively. In all experiments, the file size was set to 80K (with read/write operations accessing 20K of data from each file). The experimental parameters chosen showed the tendencies between models very clearly. The uniformity over all experimental set-ups was chosen because of representational simplicity not because different parameter ranges exhibited different tendencies. The **performance measurements** (as in prior experiments) were based on the percentage of task failures during a given time interval (1 second), and equally on **survivability**. *Each experiment was run 10 times with results averaged*. Beyond this we did up to 30 runs for better judgment on survivability but found no significant differences compared to the smaller number of runs which is then the basis of the report. In addition, survivability was measured in the following way: if an essentially critical task failed during one of the experimental runs then all other runs were counted as failed at the same failure time at which the one run had experienced the failure.

Using the above task profile parameters the following three distinct task profiles were developed:
Read Dominance: In this task profile there was a significant dominance (75%) of reading tasks,
Even Mix: This profile contains an even distribution of read/write operations (50%/50%),
Write Dominance: In this profile there was a significant dominance (75%) of writing tasks.
Criticality for each of the profiles was varied between the ranges of 6-9, 9-12 and 12-15. Sensitivity for each of the profiles was varied between the ranges of 6-9, 9-12 and 12-15. This resulted in 9 distinct sets of experiments for each of the three task profiles. In this manner the results clearly showed the influence criticality, sensitivity and read/write dominance on the deadline failure rate performance and survivability performance of the three models (Public, Private and MELODY).

Task Parameter	Range of Values	
	Read Access	Write Access
Deadline	12..15 msec	70..95 msec
Worst Case Execution Time	5 msec	10 msec
Next Arrival Time	230..330 msec	350..450 msec
Criticality (C_j)	varied [5..11], [7..13], [9..15]	
Sensitivity (R_j)	varied [6..9], [9..12], [12..15]	
# of Required Files	2-4	
Read / Write Dominance	Write Dominance: 25% readers / 75% writers Even Mix: 50% readers / 50% writers Read Dominance: 75% readers / 25% writers	

Table 8-1: File System Experiments Task Profile Parameters

8.1 Read Dominance

In an environment where the dominance of read operations is high, the urgency to have a local file

copy increases due to tendency of read operations to require fast access to file copies. In this situation, the increased overhead resulting from a public copy would be at a minimum (due to the infrequent update operations). The primary influence on file access response time performance is whether the file is available at the local site. Therefore, it's expected (and shown in previous simulations) that as the number of sensitive read operations increase the Public model is able to improve its deadline failure rate performance. The improved performance resulted directly from the model's ability to replicate or relocate public copies to those sites requiring local read access. The opposite relation is true for the Private model. Only as the number of robust read operations increase does the deadline failure rate performance of the Private model improve, since only Robust task instances are able to utilize a Private file copy. Under conditions where there is an increased number of essentially critical task instances, the Private model also improves its deadline failure rate performance (essentially critical task instances use either a shadow or private copy). However, more important is the improved survivability performance as the Private model is able to respond quicker to creating a private copy at a site where a nearly essentially critical task instance has been detected. This improved survivability performance should be apparent throughout all ranges of sensitivity, since there is no dependency of essentially critical task instance on the sensitivity of the task instance.

In figure 8-1, there is a dominance of sensitive tasks (sensitivity range 6 to 9) in the task profile. The deadline failure rate performance clearly shows that following an initial increased overhead (time intervals 1 through 4), resulting from the replication or relocation of public copies, that the Public model is able to significantly improve its performance in respect to the Private model. This is a direct result of the increased number of task instances utilizing a local public file copy, therefore reducing the communication overhead within the system. Throughout the experiment, it's quite apparent that the MELODY model clearly outperformed the two simpler models.

In figure 8-2, an increasing number of robust tasks (sensitivity range 9 to 12) allows the Private model to significantly improve its deadline failure rate performance. This improved performance allows the Private model to even match the performance of the Public model. The MELODY model clearly improves its performance resulting from its ability to locate either a public or private copy of a file at a site depending on the sites read access requirements.

In figure 8-3, the improved performance of the Private model is most dramatic, resulting in its deadline failure rate performance surpassing that of the Public model. its performance increases so dramatically, that during the initial 5 seconds the Private model's performance surpasses that of the MELODY model. However, the MELODY model is able to improve its performance much more than the Private model by correctly locating public and private copies at those sites that most require a certain type of file copy. This ability to dynamically locate either a public or private file copy, depending on task access requirements, has clearly shown to be superior under conditions of relatively low criticality (range 9 to 15).

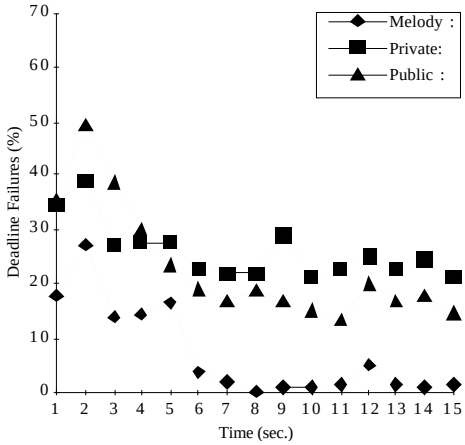


Figure 8-1: Read Dominance (C_j[9..15]/R_j[6..9])

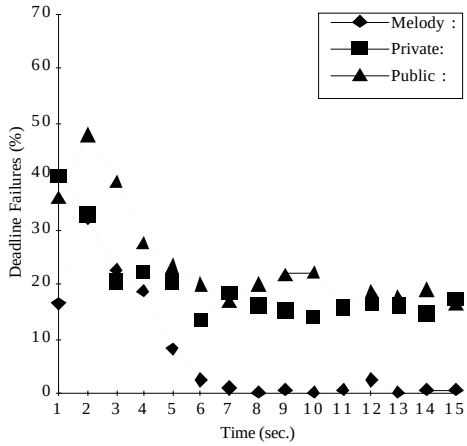


Figure 8-2: Read Dominance (C_j[9..15]/R_j[9..12])

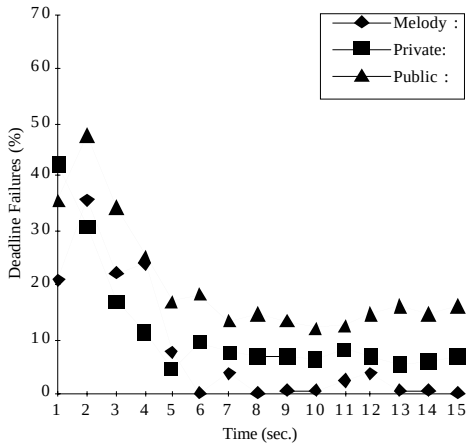


Figure 8-3: Read Dominance (C_j[9..15]/R_j[12..15])

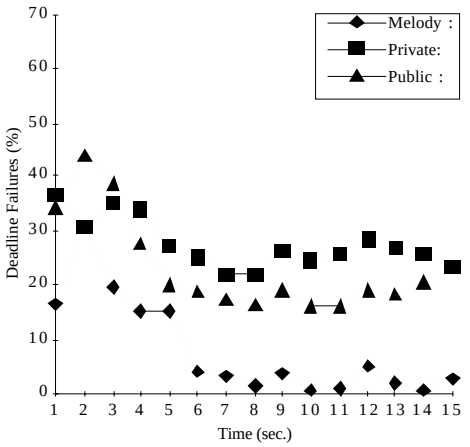


Figure 8-4: Read Dominance (C_j[7..13]/R_j[6..9])

As the criticality of the task profile is increased (range 7 to 13) clearer tendencies regarding the survivability of the models become apparent. In a middle range of criticality (figure 8-4, figure 8-5 and figure 8-6) the characteristic tendency in regards to the Private model's performance improvement (under increasing levels of robust task instances) is still quite apparent (figure 8-6). The increased criticality causes significant problems for the Public model as it's no longer to complete the entire experiment (failing between the 10th and 14th second in figure 8-6). However, the Private model is able to survive, and shows the clear benefit in being able to create a private copy at a site where a nearly essentially critical task instance has been detected. It's quite apparent that this ability to handle essentially critical task instance is a significant benefit of private copies. The MELODY model throughout all three experiments was clearly able to handle the increased level of criticality, while still outperforming the other two simpler models in respect to deadline failure rate performance.

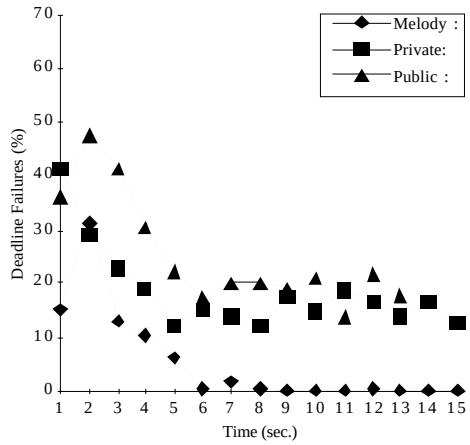


Figure 8-5: Read Dominance (C_j[7..13]/R_j[9..12])

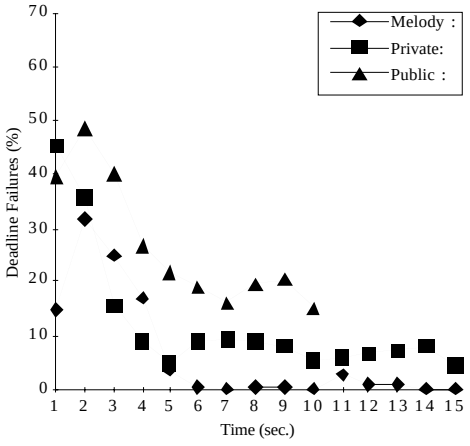


Figure 8-6: Read Dominance (C_j[7..13]/R_j[12..15])

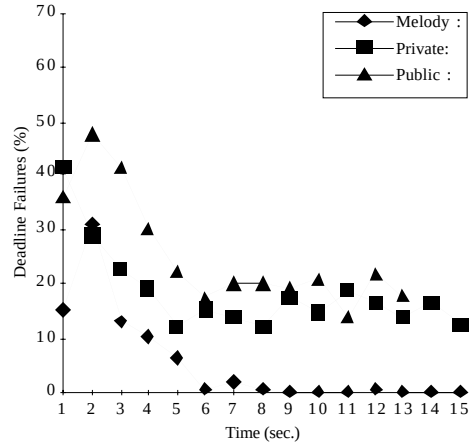


Figure 8-7: Read Dominance (C_j[5..11]/R_j[6..9])

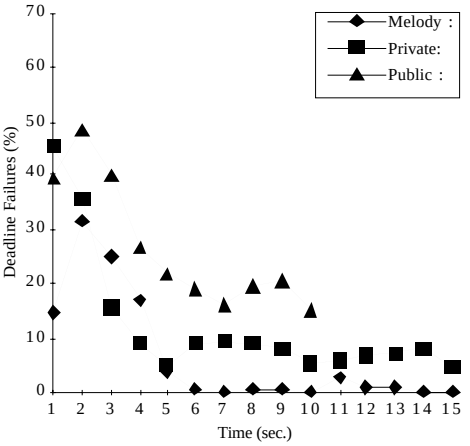


Figure 8-8: Read Dominance (C_j[5..11]/R_j[9..12])

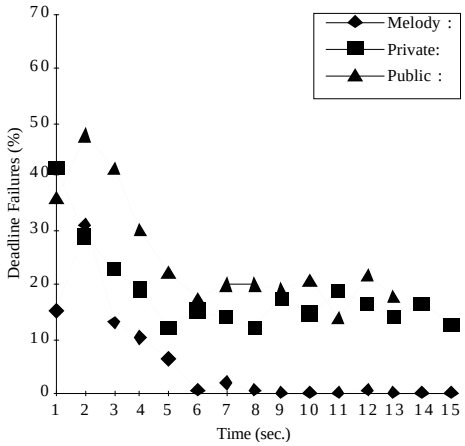


Figure 8-9: Read Dominance (C_j[5..11]/R_j[12..15])

A significant increase in the criticality of the task profile (range 5 to 11) caused even more problems for the Public model. Under this high level or criticality, the Public model is only able to survive up to 8 seconds under all ranges of sensitivity (figure 8-7, figure 8-8 and figure 8-9). The Private model also is no longer able to survive for the entire experimental time (failing in the 13th second). However, a significant increase in the performance of the Private model can be seen in figure 8-9. Under this high level of criticality, and increased number or essentially critical task instances, the Private model's deadline failure rate performance exceeds that of the Public model. This improved deadline failure rate results in the model's improved performance in respect to survivability. The MELODY model again is able to dynamically adjust the number and location of both public and private copies to the point that it clearly shows a superior performance to the two simpler models.

8.2 Even Mix

In environments where there is an even distribution of read/write operations the it was expected that the (and shown in previous simulations) problem resulting from creating additional public copies becomes more apparent. This problem results from the fact the write access to a public copy is directly linked to the number of copies located within the system, and has a direct influence on the ability of the task instance to complete prior to its associated deadline. However, reading task instances still prefer to have a local file copy to improve the chances that they meet their deadlines. These conflicting requirements require that the models correctly handle not only the number of public or private copies, but also the location of those copies. The characteristic tendency in respect to the deadline failure rate performance of the Public model and Private model under varying degrees of sensitivity should still be apparent (only to a lesser degree due to the smaller number of task instances benefiting from either a public or private copy).

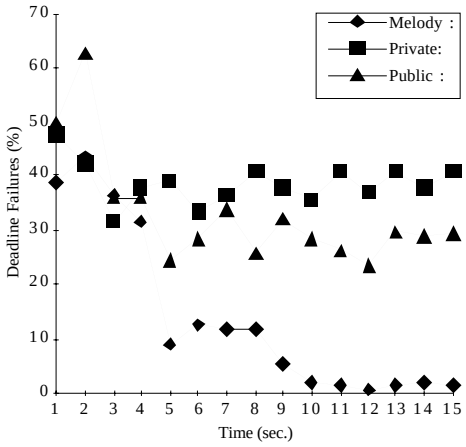


Figure 8-10: Even Mix (C_j [9..15]/ R_j [6..9])

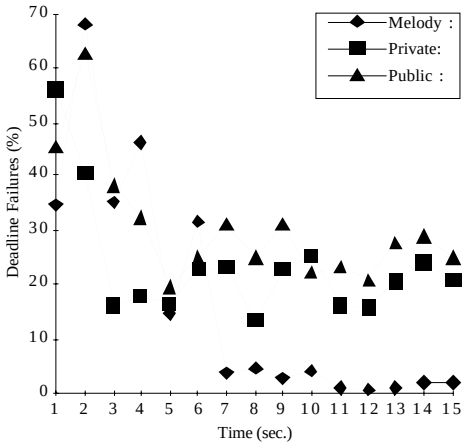


Figure 8-11: Even Mix (C_j [9..15]/R_j [9..12])

Under a varying range of sensitivity (figure 8-10, figure 8-11 and figure 8-12) the Private model is still able to improve its performance in respect to the Public model. The performance of all three models is significantly degraded due to the increased number of writing task instances, therefore and increased level of competition for those writing task instances. Under this low level of criticality (all task instances nearly non-critical) all models are able to survive the entire experimental run.

As the level of criticality is increased (range 7 to 13) the tendency of the Public model to fail in terms of survivability is also still apparent under the increased number of writing tasks. The Public model does however survive for a longer period of time (failing in either the 12th or 13th seconds). Both the Private model and MELODY model are still able to handle this level of criticality and survive for the entire experimental run, as in the Read Dominance experiments.

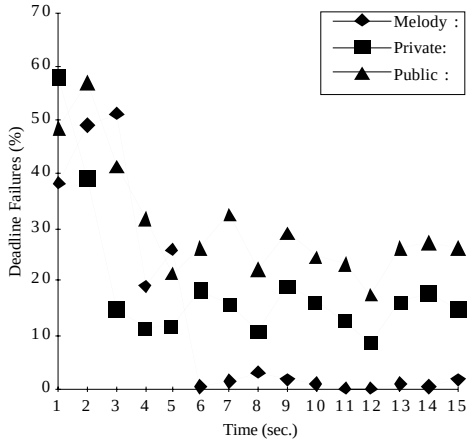


Figure 8-12: Even Mix (C_j [9..15]/R_j [12..15])

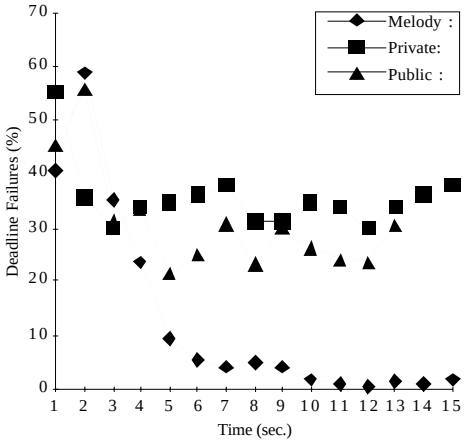


Figure 8-13: Even Mix (C_j [7..13] / R_j [6..9])

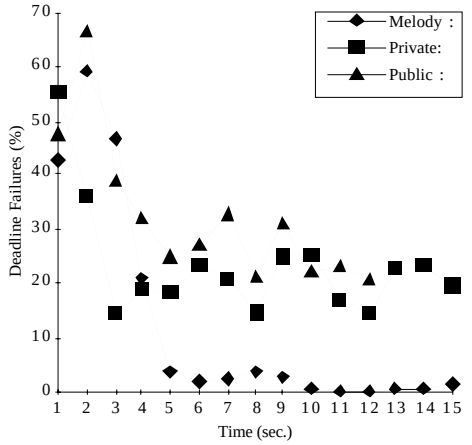


Figure 8-14: Even Mix (C_j [7..13] / R_j [9..12])

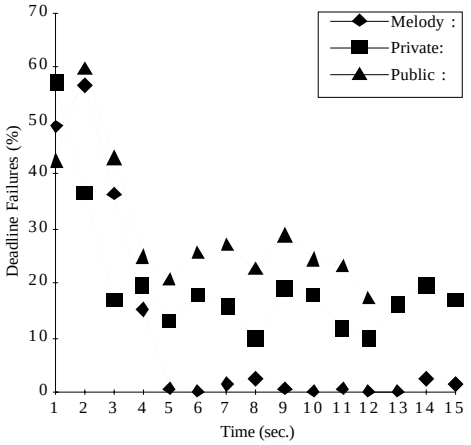


Figure 8-15: Even Mix (C_j [7..13]/R_j [12..15])

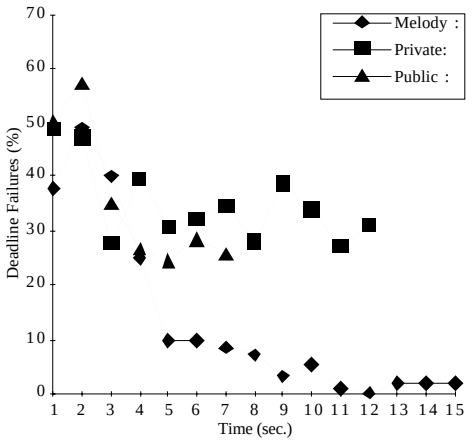


Figure 8-16: Even Mix (C_j [5..11] / R_j [6..9])

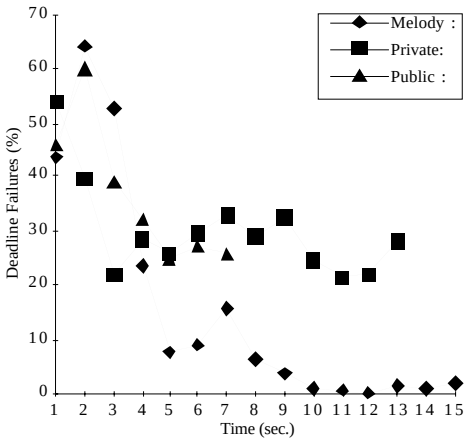


Figure 8-17: Even Mix (C_j [5..11] / R_j [9..12])

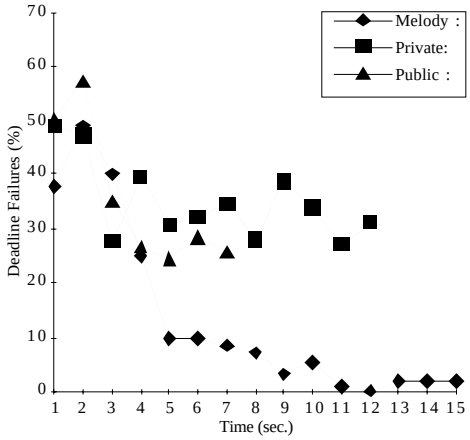


Figure 8-18: Even Mix (C_j [5..11] / R_j [6..9])

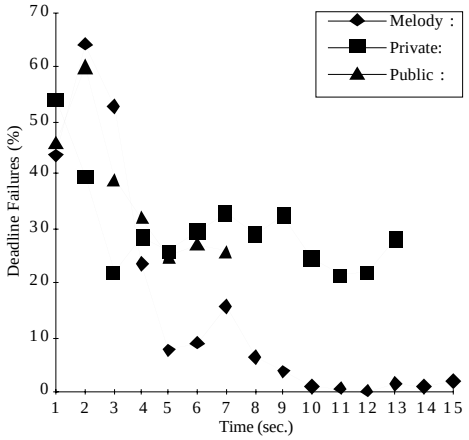


Figure 8-19: Even Mix (C_j [5..11] / R_j [9..12])

A significant change in the performance of the Private model can be seen as the level of criticality is increased (range 5 to 11) (figure 8-18, figure 8-19 and figure 8-20). The improved performance of the Private model under high levels of criticality and high levels of sensitivity (range 6 to 9) is no longer superior to the Public model's deadline failure rate performance (figure 8-18) as was apparent in the Read Dominance experiments (figure 8-8). However, the Private model's performance is still superior where a majority of task instances are Robust (figure 8-20). The performance of the MELODY model both in terms of deadline failure rate and survivability is still significantly apparent under all levels of criticality and sensitivity.

A significant change in the performance of the Private model can be seen as the level of criticality is increased (range 5 to 11) (figure 8-18, figure 8-19 and figure 8-20). The improved performance of the Private model under high levels of criticality and high levels of sensitivity (range 6 to 9) is no longer superior to the Public model's deadline failure rate performance (figure 8-18) as was apparent in the Read Dominance Experiments (figure 8-8). However, the Private model's performance is still superior where a majority of task instances are Robust (figure 8-20). The performance of the MELODY model both in terms of deadline failure rate and survivability is still significantly apparent under all levels of criticality and sensitivity.

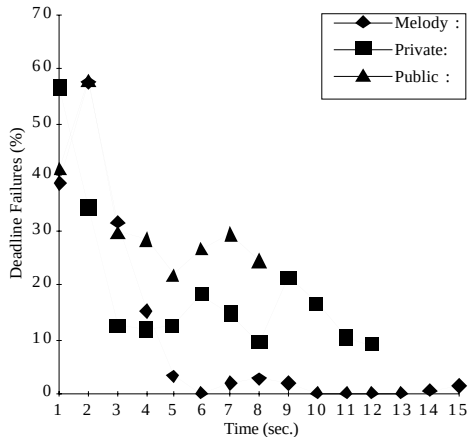


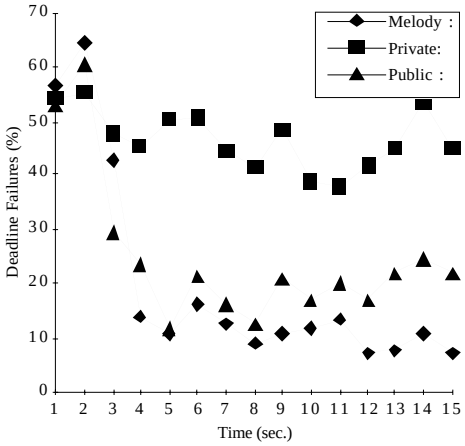
Figure 8-20: Even Mix (C_j [5..11]/R_j [12..15])

8.3 Write Dominance

In environments were there is a dominance (75%) of write operation it was expected (and shown in previous simulations) that the problem resulting from creating additional public copies becomes much more apparent then in the Even Mix Experiments. This results from the fact that there is very little benefit in creating any additional public copies (due to the significant overhead). Private copies are also no longer significantly beneficial due to the frequency of update operations and the overhead

this would cause to maintain a private copy at a site. This results in the models having to correctly locate the available public copies (if possible), control the number of public or private copies created, and correctly utilize private copies (if possible) to improve performance both in respect to the deadline failure rate and more importantly the survivability.

Under a varying range of sensitivity (figure 8-21, figure 8-22 and figure 8-23) the Private model's deadline failure rate performance no longer surpasses the performance of the Public model. This significant change in the performance of the Private model results directly from the lack of any significant benefit from private copies. Only under conditions where there is a high level of Robust task instances (range 12 to 15) (figure 8-23) does the Private model show an improved deadline failure rate performance. However, this performance does not at any point surpass the performance of the Public model.



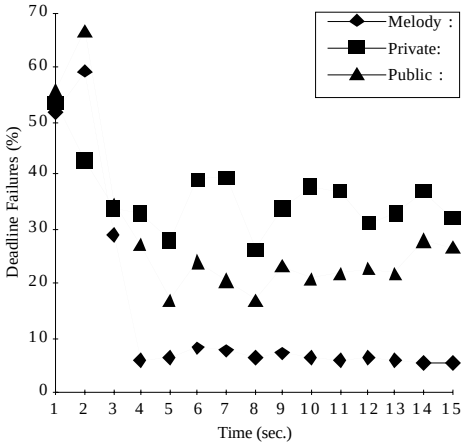
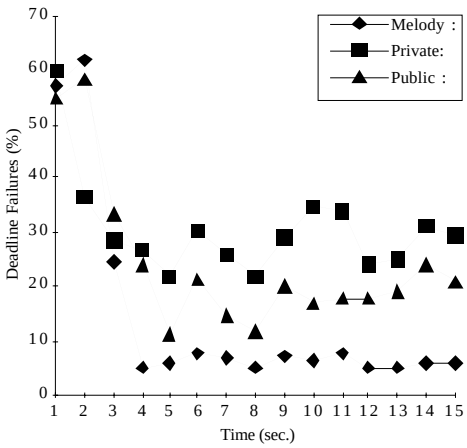


Figure 8-21: Write Dominance (C_j[9..15]/R_j[6..9])**Figure 8-22:** Write Dominance (C_j[9..15]/R_j[9..12])

As the level of criticality in the task profile increases (figure 8-24, figure 8-25 and figure 8-26), a significant change in the performance of the Public model is apparent. The failures experienced by this model in the Read Dominance and Even Mix profiles no longer occur. Resulting from the model's ability to reduce the number of public copies, which results in a greater chance for writing task instances to survive. This reduction in the number of public copies created does not significantly affect the reading task instances due to a smaller communication overhead, which results in increased response time in remote access. Performance of the MELODY model is significantly worse than in the Read Dominance and Even Mix experiments, but still is able to survive for the entire experimental run and surpass the performance of the other two simpler models.



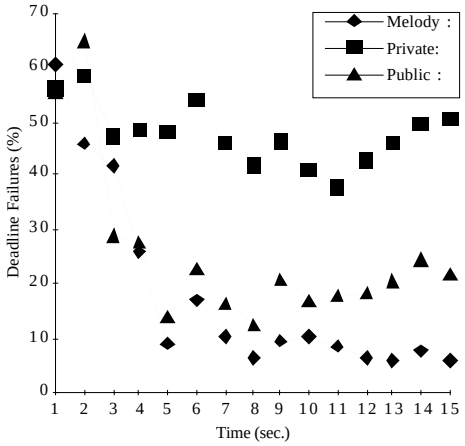
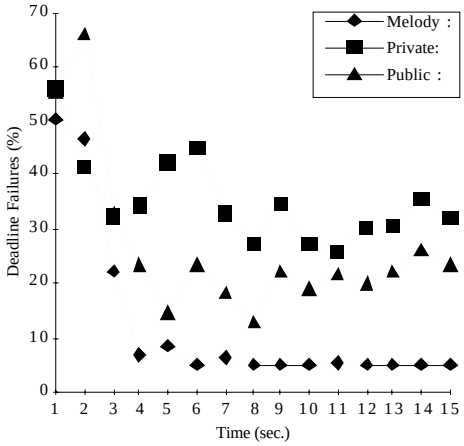


Figure 8-23: Write Dominance (C_j[9.15]/R_j[12.15]) **Figure 8-24:** Write Dominance (C_j[7..13]/R_j[6..9])

The change in performance of the Private model, under high levels of criticality (range 5 to 11), is still the case as the number of writing task in the profile is increased (figure 8-27, figure 8-28 and

figure 8-29). However, this improved deadline failure rate performance is not as apparent due to the early failure of the Private model, and never appears to surpass the performance of the Public model. Both the Public and Private models fail very early, while the MELODY model is able to survive the entire experimental run (under all ranges of sensitivity).



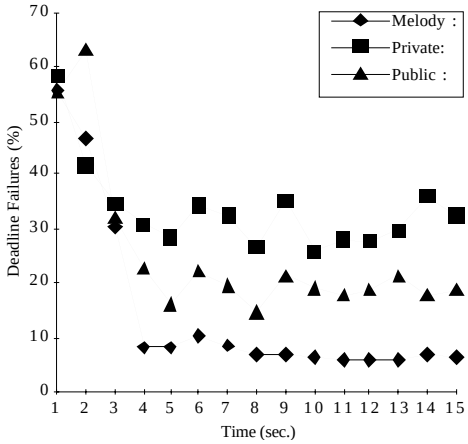
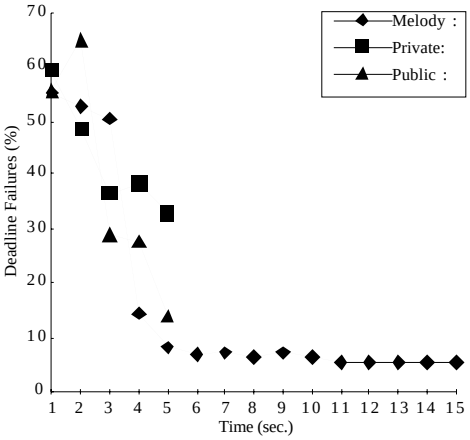


Figure 8-25: Write Dominance (C_j[7..13]/R_j[9..12]) **Figure 8-26:** Write Dominance (C_j[7.13]/R_j[12.15])

The MELODY model's ability to improve its performance, both with respect to deadline failure rate and survivability, clearly showed a significant benefit in unpredictable and safety-critical environments. Its ability to dynamically relocate, replicate and delete public copies while creating and deleting private copies, as necessary, clearly outweighs any overhead that may result from the increased complexity of the model. This pay-off for safety-critical systems, requiring adaptive measures to operate in unpredictable environments, is even more significant since criticality and sensitivity have such a strong deteriorating influence on both the Public and Private models.



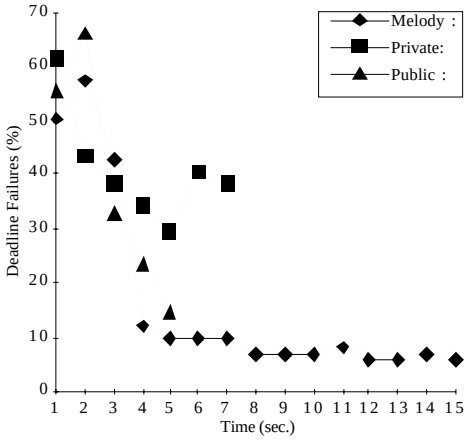


Figure 8-27: Write Dominance (C_j[5..11]/R_j[6..9]) **Figure 8-28:** Write Dominance (C_j[5..11]/R_j[9..12])

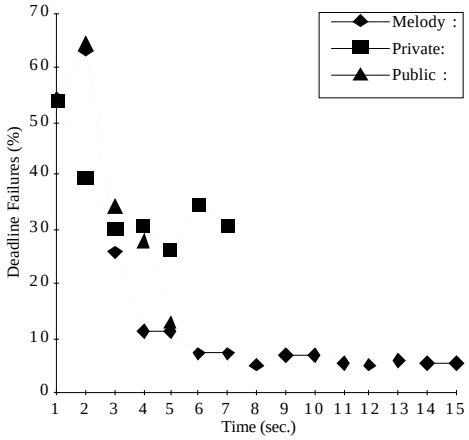


Figure 8-29: Write Dominance (C_j[5.11]/R_j[12.15])

From these results, it was apparent that the distributed experiments closely reflected the results obtained under simulation, both in terms of deadline failure rate and survivability. Throughout all of the charts it can be seen that the unpredictability of the environment caused significant changes in the performance during the experiments (resulting in deadline failure rate variations between 5 and 10 percent). This performance variation directly resulted from delays in communication, computation and file manipulation within a distributed environment. These variations were not seen in the previous simulation results [WeK93] due to the strict control on the times set for these activities. For example, in the simulation result the variation after the first few seconds was no more than 1 to 2 percent. Delays in communication times were randomly selected between the range of 3 to 5, but never exceeded this range. Computation time and file manipulation times were simulated to adhere very tightly to the times modeled in the task profiles. For the distributed experiments this was different. Communication times could exceed the expected bounds (under high communication loads), while computation and file manipulation times were dependent on the number of records accessed and the load on the processor. However, at no point of time was there any significant deviation from characteristic tendencies observed between the three models.

Chapter 9 File Server/Task Scheduler Integration Experiments

As discussed in section 1.2, since the purely local task scheduling activity takes much less time than remote file acquisition integration of task and resource scheduling means most likely to invoke the Task Scheduler during the resource scheduling activities of the File Server. Integration of task scheduling activities would have to handle the following conflicting goals:

- Infrequent invocation of the Task Scheduler. This minimizes the overhead caused by Task Scheduler activities (increasing the time for resource scheduling activities) while at the same time the number of task instances waiting to be scheduled is increased thus providing for 'wiser' scheduling decisions. This policy equally means to invoke the Task Scheduler *as late as possible*.
- Invocation of the Task Scheduler *as early as possible*. This increases, for every individual task instance, the time available for acquiring the needed resources.

Delaying the invocation of the Task Scheduler for a longer time span improves the Task Scheduler's scheduling process by providing a wider basis of decision, however this increased delay in a task instance's task scheduling phase (see section 3.3) could also have significant effects on a task instance's ability to complete before its associated deadline by reducing the time allowed to complete the remaining task life cycle phases. More important than the increased deadline failure rate resulting from poor Task Scheduler/File Server integration is the possible effect that this integration policy would have upon the survivability of the MELODY system. As a result, we have developed and extensively investigated the following integration models:

Periodic model: In the Periodic model the Task Scheduler is invoked after a predetermined interval of time has passed. This model is denoted as *Periodic (<period>)*.

Dynamic model: In the Dynamic model the following three dynamic thresholds are used to determine when to invoke the TS:

- Threshold **FTh** is set for the File Server competing task queue, and monitors the number of task instances currently competing for access to shared resources located either locally or remotely.
- Thresholds **STh1** and **STh2** (where **STh1** ≤ **STh2**) are set for the TS waiting queue. These thresholds monitor the number of task instances currently waiting to be scheduled by the Task Scheduler at the local site. These thresholds however do not take into account the number of essentially critical task instances since essentially critical task instances are immediately placed into the File Server's competing task queue (essentially critical task instances are not required to be scheduled by the Task Scheduler due to their high criticality).

The Dynamic model would then invokes the Task Scheduler whenever either of the following conditions becomes true:

- **FTh** has been underpassed (the number of task instances at the current site competing for shared resource has fallen below a predetermined value), and **STh1** has been surpassed (there are enough task instances waiting to be scheduled that TS can create a fairly good task schedule),
- **STh2** alone has been surpassed (there are enough task instances waiting to be scheduled by TS that it should be invoked immediately regardless of the current level of competition at the current site).

The threshold values are conceived to be adaptable, over certain time intervals, to specific characteristics of the profiles encountered.This model will be denoted by *Dynamic(-|FTh,STh1, STh2)*.

Under extreme settings of the threshold values for the Dynamic model invocation of the Task Scheduler could be too frequent. Therefore, it determined that it may be beneficial to delay invocation for a certain interval of time as defined in the Periodic model.

Adjusted model: This model adjusts the Dynamic model by only invoking the Task Scheduler at certain predefined instances in time. The thresholds **FTh**, **STh1** and **STh2** (as defined in the Dynamic model) are used to determine when to invoke the Task Scheduler. This model however delays invocation until the next period (defined in the Periodic model). This model will be denoted by *Adjusted (<period> | FTh, STh1, STh2).*

In this way the characteristic tendencies of the Adjusted model could be directly attributed to the either the Dynamic model (under very short periods), the Periodic model (under very long periods), or a combined influence from both models (this will be discussed in detail in the following experimental results).

The main expectation for the new integrated task and resource scheduling policies is that the lack of guarantee by the Task Scheduler would be more than made up by the less chaotic and harmful way of early resource locking (as used in traditional operating system design). Our first concern was to compare the new integration policies with a classical model of resource scheduling *prior* to task scheduling. This was subject to extensive simulation experiments reported in [WeL94]. Our main concern in the following experiments is not the deadline failure rate performance of the models in respect to the classical model, but more importantly the performance of the models in respect to each other under varying degrees of criticality (***survivability***). As a result the following three distinct task profiles have been developed:

Low Criticality: In a low criticality task profile a dominance of tasks whose criticality was nearly non-critical was established (criticality in the range of [6..9]). This profile would be used to compare the deadline failure rate performance of the models to previous well understood simulated experimental results,

Middle Criticality: This profile contained a slightly more critical set of tasks whose criticality ranged between the values of [4..7],

High Criticality: In this profile there was a significant dominance of tasks whose criticality was nearly essentially critical (criticality in the range of [2..5]).

Sensitivity for each of the profiles was varied between the ranges of 4-9 (all tasks being sensitive initially). For each of the criticality ranges we varied the characteristic parameters <period>, FTh, STh1, and STh2 as shown in table 9-1:

Integration Parameter	Range of Values
<period>	varied 10, 50, and 90
File Server Threshold FTh	varied 1, 3, and 5
Task Scheduler Threshold STh1	varied 1, 3, and 5
Task Scheduler Threshold STh2	varied 1, 3, and 5

Table 9-1: Task Scheduler/File Server Integration Experiment Parameters

This resulted in 54 distinct sets of experiments for each of the three task profiles. In this manner the results clearly showed the influence of criticality in respect to the setting of the <period>, FTh, STh1, and STh2. Performance of the three models (Periodic, Dynamic and Adjusted) was measured both in respect to the deadline failure rate performance (percentage of task instances failing during a given interval of time) and in respect to survivability performance (at which point in time did the first essentially critical task instance fail).

In each of the experiments discussed here, 7 sites were used with 100 task distributed among all sites. 16 data files were distributed among all sites. There were initially three public copies and no private copies for each data file. Parameter ranges for the task profiles (from which initial settings for the tasks in the profile were made) are shown in table 9-2. In the following task profiles, task deadlines had then been modeled accordingly to fit tightly. The rather high deadline failure rates (which can be seen in the following figures) were thus to be expected. They are not unacceptable as long as survivability is guaranteed, i.e. the failed task instances had not yet been essentially critical. In this way characteristic differences in the performance of the models under a relatively high load could be more accurately determined.

Task Parameter	Range of Values	
	Read Access	Write Access
Deadline	10..20 msec	40..50 msec
Worst Case Execution Time	3 msec	5 msec
Next Arrival Time	50..90 msec	150..290 msec
Criticality (C_j)	varied [2..5], [4..7], [6..9]	
Sensitivity (R_j)	sensitive ranging between [4..9]	
# of Required Files	2-4	
Read / Write Dominance	75% readers / 25% writers	

Table 9-2: Task Scheduler/File Server Integration Experiments Task Profile Parameters

The threshold values for criticality a_i' and a_i" were uniformly set to 1 and 10, respectively, while threshold values for sensitivity b_i' and b_i" were uniformly set to 5 and 10, respectively, across all sites. In all experiments the file size was set to 80K, with both read/write operations accessing 20K of data from each file requested. The experimental parameters chosen showed the tendencies between models very clearly. The uniformity over all experimental set-ups was chosen because of representational simplicity not because different parameter ranges exhibited different tendencies. *Each experiment was run 10 times with results averaged*. Beyond this we did up to 30 runs for better judgment on survivability but found no significant differences compared to the smaller number of runs which is then the basis of this report.

9.1 Low Criticality

Initial experiments were conducted to determine the models' performance strictly in respect to the deadline failure rate performance of the integration. The experiments were set up using a task profile that had a dominance of low criticality tasks (nearly non-critical). Under these conditions, it was expected (from the results obtained by the simulator) that the performance of the Periodic model would have an optimal setting with respect to the deadline failure rate. This performance would then degrade as the period was increased or decreased from the optimum. For the Dynamic model there should also be optimal settings for FTh, STh1 and STh2. Also, the Adjusted model had been developed to assure that under certain extreme settings its performance would match one of its constituent models. For example, a very long period for the Adjusted model would cause it to be significantly influenced by the period, and as a result its performance should be nearly identical to the Periodic model. While, a very short period should match the performance of the Dynamic model. Using these basic functional expectations, other features of the Adjusted model could then securely be evaluated, and separated from the effects discussed.

The performance of the Periodic model in respect to a varying interval can be seen in figure 9-1, figure 9-2 and figure 9-3. The Periodic model's performance is optimal for a period of 50 (figure 9-2). However, its performance degrades significantly as the period is increased or decreased (figure 9-1 and figure 9-3). This optimal interval is directly related to the task profile selected as was found by studying various profiles, and can be seen throughout all of the following charts.

Since the interval is not utilized as part of the Dynamic model, its performance is unchanged in figure 9-1, figure 9-2 and figure 9-3. In figure 9-4, figure 9-5 and figure 9-6, threshold FTh is varied from 1 to 5, showing an optimal setting of FTh to be 3 (figure 9-5). As FTh is increased or decreased the model's performance degrades significantly (resulting from either the infrequent (figure 9-4), or more frequent (figure 9-6) invocation of TS). In figure 9-6, figure 9-9 and figure 9-10, the influence of STh1 on the Dynamic model's performance is shown as the threshold STh1 is varied from 1 to 5. These figures clearly show a degraded performance as STh1 is increased to 5 (figure 9-10), resulting from a severely delayed invocation of TS. This performance degradation is so sever that the Dynamic model's performance matches the poor performance of the Periodic model. The variation of STh2 (between the ranges of 1 and 5) is shown in figure 9-10, figure 9-7 and figure 9-8. Setting STh2 to either 3 or 5, shows significant performance improvement to a point that in both charts the Dynamic model again performs better that the Periodic model (figure 9-8).

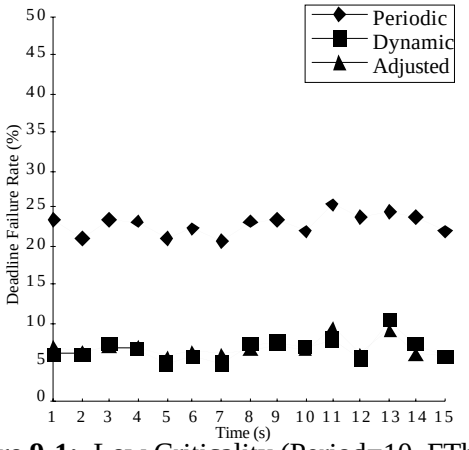


Figure 9-1: Low Criticality (Period=10, FTh=1, STh1=1, STh2=1)

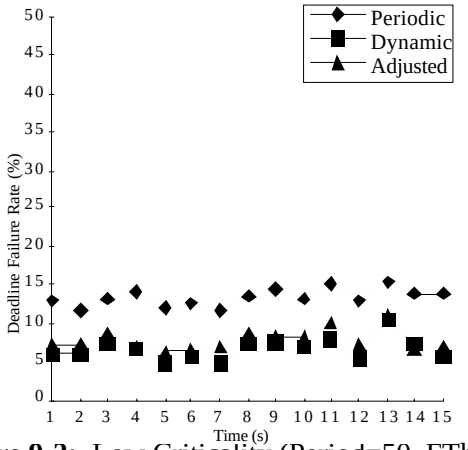


Figure 9-2: Low Criticality (Period=50, FTh=1, STh1=1, STh2=1)

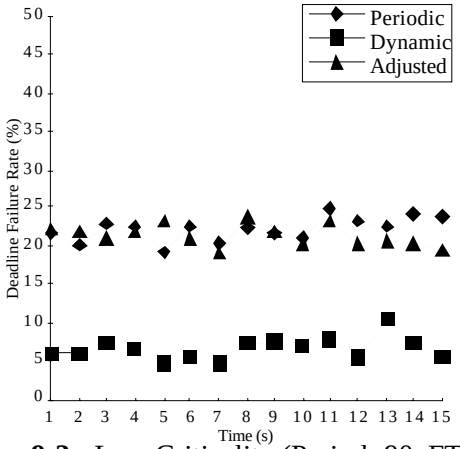


Figure 9-3: Low Criticality (Period=90, FTh=1, STh1=1, STh2=1)

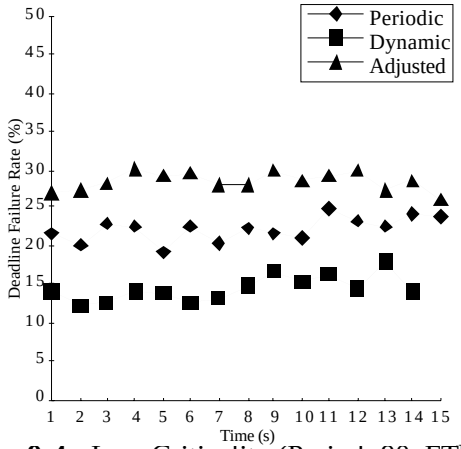


Figure 9-4: Low Criticality (Period=90, FTh=1, STh1=1, STh2=5)

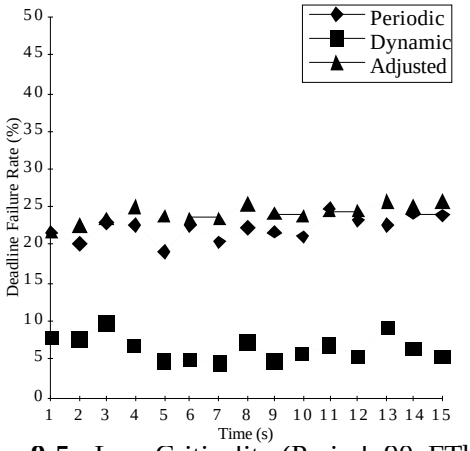


Figure 9-5: Low Criticality (Period=90, FTh=3, STh1=1, STh2=5)

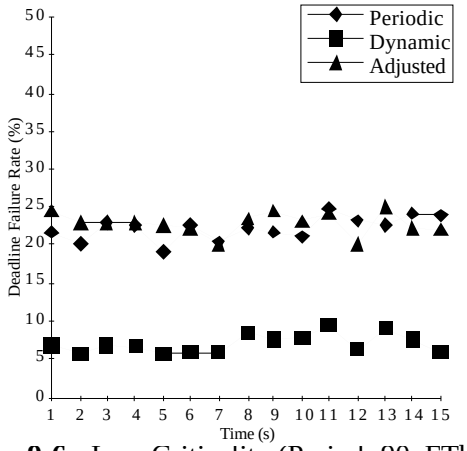


Figure 9-6: Low Criticality (Period=90, FTh=5, STh1=1, STh2=5)

The effect of the interval on the performance of the Adjusted model can be seen in figure 9-1, figure 9-2 and figure 9-3. When the interval was set to 90 (figure 9-3) this caused the Adjusted model's performance to match that of the Periodic model. Shorter intervals caused the Adjusted model's performance to match more closely the performance of the Dynamic model (figure 9-1). This result from the fact that a short interval causes the Adjusted model only to slightly delay the invocation of TS in comparison to the invocation of the Dynamic model. However, this is only a result under condition where the Dynamic model is invoking the TS quite frequently (more frequently than the Periodic model). If the Dynamic model is invoking TS less frequently than the Periodic model, the short interval causes an additional delay in the invocation of TS that either improves or degrades the Adjusted

model's performance. This degraded performance due to additional delay caused by the interval can be seen in figure 9-7. In all charts where the interval was set to 90 for the Adjusted model, its performance was significantly worse than that of the Dynamic model (only matched that of the Periodic model under certain parameter setting as seen in figure 9-3).

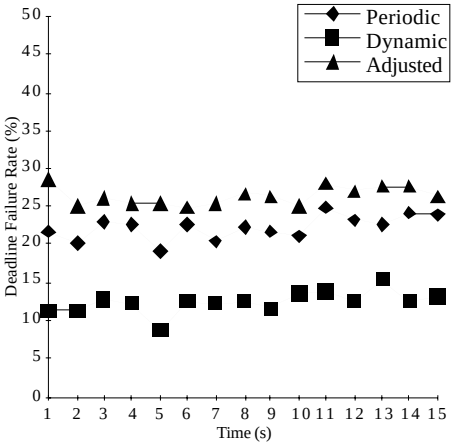


Figure 9-7: Low Criticality (Period=90, FTh=5, STh1=3, STh2=3)

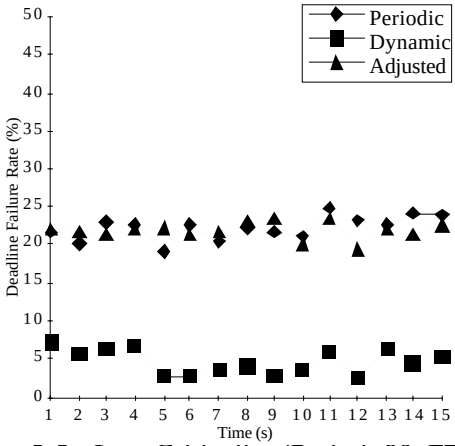


Figure 9-8: Low Criticality (Period=90, FTh=5, STh1=1, STh2=1)

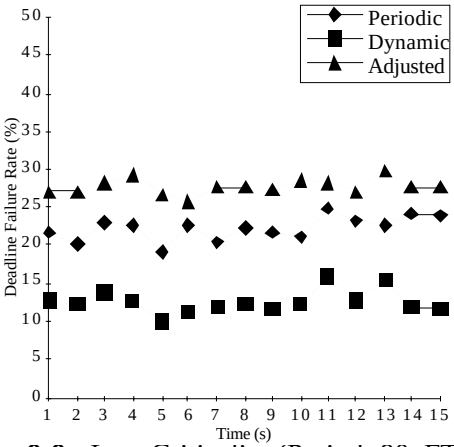


Figure 9-9: Low Criticality (Period=90, FTh=5, STh1=3, STh2=5)

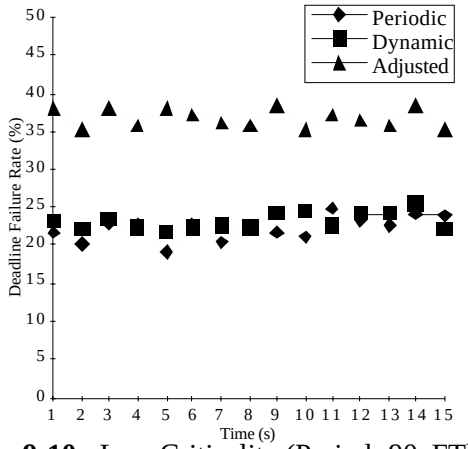


Figure 9-10: Low Criticality (Period=90, FTh=5, STh1=5, STh2=5)

Overall the Periodic model only outperformed any of the other models when its interval was set to 50 (the optimal setting), and the other models' performances were at their worst. In all other cases, the Dynamic model's performance was never the worst of all models. Throughout nearly all of the charts its performance is the better or nearly the same as one of the other models. The Adjusted model was able to match the performance of the Dynamic model under certain parameter ranges, however in cases with long periods its performance was consistently worse than the Dynamic model. The results seen here matched very closely (with no significant variation) those obtained by the simulator. As a result we then proceeded to extend our investigation of the integration methods by increasing the criticality of the task profile.

9.2 Middle Criticality

To determine the effect that an increased level of criticality has on the TS/FS integration, the criticality of the task profile was set between the ranges of 4 and 7. The initial expectations were that the characteristic tendencies observed under a low criticality task profile should be observable in a task profile with a slightly increased level of criticality. The characteristic difference is that essentially critical task instances would not be effected by the method of TS/FS integration (since they are immediately placed into the File Server's competing task queue and never scheduled by TS). However, nearly essentially critical task instances would be effect by the chosen method of TS/FS integration (and the parameters used for that integration). Improved performance, resulting from the integration

policy, should result in fewer essentially critical task instances, therefore resulting in a reduced risk of an essentially critical task instance missing its deadline. However, a degraded deadline failure rate performance should have the effect of causing increased levels of essentially critical task instances, therefore resulting in increased risk that an essentially critical task would miss its associated deadline.

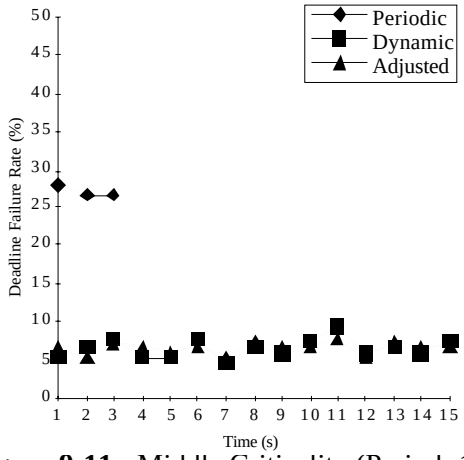


Figure 9-11: Middle Criticality (Period=10, FTh=1, STh1=1, STh2=1)

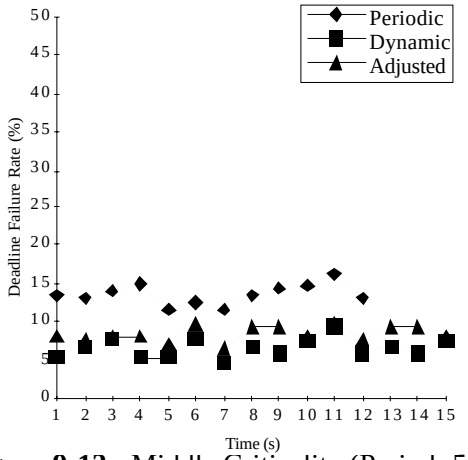


Figure 9-12: Middle Criticality (Period=50, FTh=1, STh1=1, STh2=1)

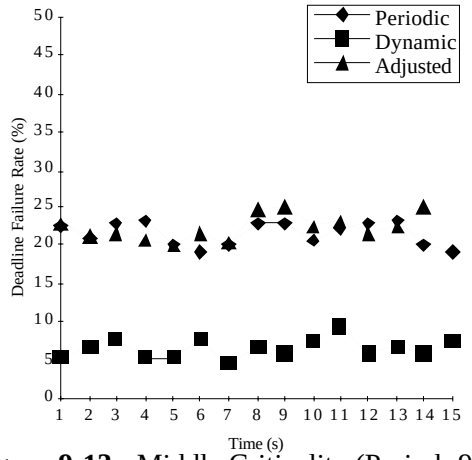


Figure 9-13: Middle Criticality (Period=90, FTh=1, STh1=1, STh2=1)

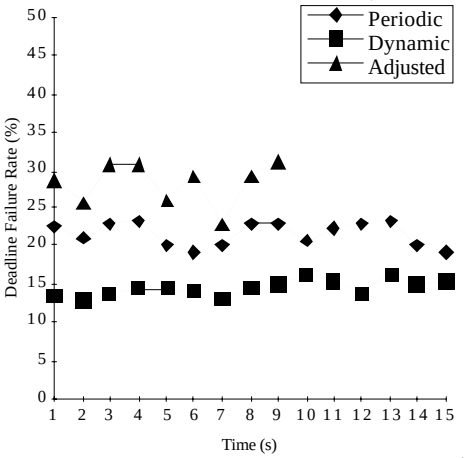


Figure 9-14: Middle Criticality (Period=90, FTh=1, STh1=1, STh2=5)

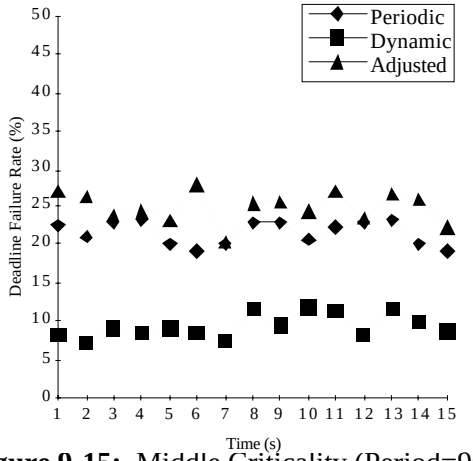


Figure 9-15: Middle Criticality (Period=90, FTh=3, STh1=1, STh2=5)

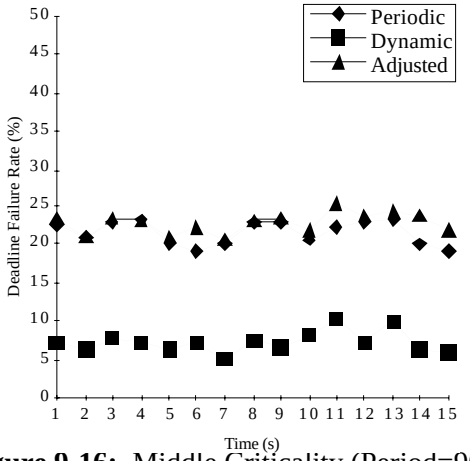


Figure 9-16: Middle Criticality (Period=90, FTh=5, STh1=1, STh2=5)

The characteristic performance of the Periodic model in respect to a varying interval was still clearly observable in all of the charts (as can be seen in figure 9-11, figure 9-12 and figure 9-13). As in the Low Criticality experiments, the Periodic model's performance shows an optimal setting for the interval 50 (figure 9-12). This performance degrades as the interval is increased or decreased from this

value (figure 9-11 and figure 9-13). However, the Periodic model is only able to complete the entire experimental run when the interval is set to 90 (figure 9-13). A short interval caused the Periodic model to fail after 3 seconds. These failures result from the increased overhead caused by the frequent invocation of the Task Scheduler. While, under long intervals the performance of the model is poorer the survivability increases to a point that the model is able to survive and complete the experiment. This results not only from the decreased overhead caused by fewer invocations of TS, but also from less competition for shared resources due to the increased number of task instances not being successfully scheduled by TS (since their deadline had expired before TS was invoked). This characteristic tendency of the Periodic model to fail under short interval was apparent in all of the charts.

As was shown in the Low Criticality experiments, the performance of the Dynamic model was not effected by the interval chosen. Its performance matched those characteristics observed under Low Criticality, with only slight changes in deadline failure rate due to increased number of essentially critical task instances. However, in none of the charts did this slightly increased deadline failure rate change the relative position of the Dynamic model in respect to the other Periodic model and Adjusted model. Also, at no point did this increased level of criticality cause an essentially critical task instance to miss its deadline.

The performance of the Adjusted model in respect to deadline failure rate clearly showed the same tendencies as in the Low Criticality experiments. In respect to the survivability of the models, the Adjusted model was able to survive the entire experimental run when the interval was set to 10 (figure 9-11). This resulted directly from its characteristic matching the Dynamic model's performance under short intervals. As the interval was increased, this model showed problems in its ability to complete the entire experimental run. With the interval set to 90 (figure 9-13) the Adjusted model was no longer able to complete the entire experimental run and failed after 14 seconds. This failure directly resulted from the model's delayed invocation of TS, and the resulting increased number of essentially critical task instances. This characteristic failure under long intervals could clearly be seen in figure 9-14, figure 9-17, figure 9-18 and figure 9-19.

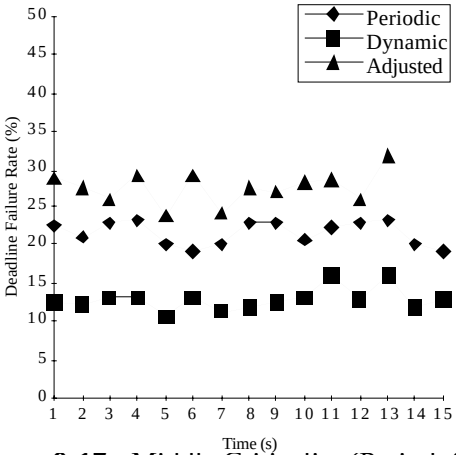


Figure 9-17: Middle Criticality (Period=90, FTh=5, STh1=3, STh2=5)

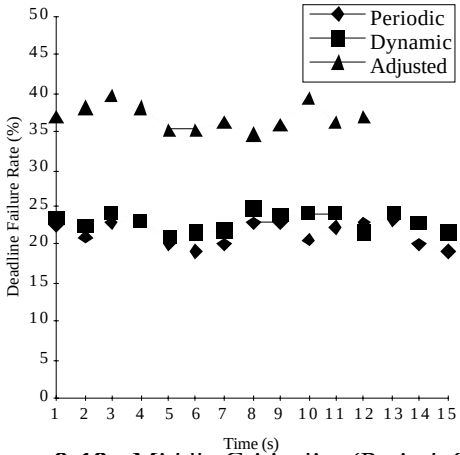


Figure 9-18: Middle Criticality (Period=90, FTh=5, STh1=5, STh2=5)

Overall the Periodic model was no longer successfully able to handle the increased levels of criticality (only completing an entire experimental run when the interval was set to 90). This survivability performance shows a significant flaw in using the Periodic model. It was only able outperformed the other models when its interval was set to 50 (the optimal setting) and the other models performances were at their worst. In all other cases, the Dynamic model's performance was never the worst of all models. Throughout nearly all of the charts its performance is better or nearly the same as one of the other models. The Adjusted model was able to match the performance of the Dynamic model under certain parameter ranges, however in cases with long periods its performance was consistently worse than the Dynamic model, even failing when the period was very long.

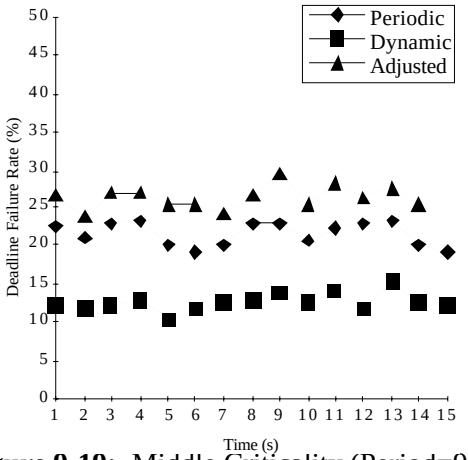


Figure 9-19: Middle Criticality (Period=90, FTh=5, STh1=3, STh2=3)

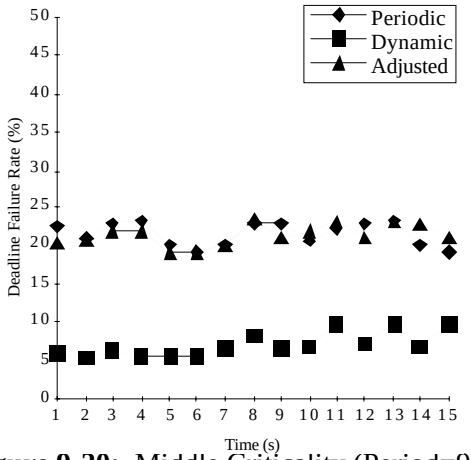


Figure 9-20: Middle Criticality (Period=90, FTh=5, STh1=1, STh2=1)

9.3 High Criticality

The effect that increasing the level of criticality such that almost all task instances are nearly essentially critical can be seen in the following experiments were the criticality of the task profile was set between the ranges of 2 and 5. As in the middle criticality experiments it was expected those characteristic tendencies observed under a low criticality task profile should be observable in a task profile with a very high level of criticality. However, the characteristic difference of essentially critical task instance not being effected by the TS/FS integration would play a more important role in effecting the performance of the integration policies. Improved performance, resulting from the integration policy, should have a very significant impact on the survivability of the models resulting from the reduced risk of an essentially critical task instance competing against another essentially critical task instance. However, a slightly degraded deadline failure rate performance would have a severe impact on a models deadline failure rate performance and survivability.

The characteristic performance of the Periodic model in respect to a varying interval was observable in all of the charts (as can be seen in figure 9-21, figure 9-22 and figure 9-23). As in the Low Criticality experiments, the Periodic model's performance shows an optimal setting for the interval 50 (figure 9-22) and degrades as the interval is increased or decreased from this value (figure 9-21 and figure 9-23). However, due to the Periodic model's inability to adjust to the changing environmental conditions the model is only able to survive for 5 seconds under the optimal setting of the period. The model's failure is even more drastic when the period is increased or decreased from this optimal setting where it fails after only one second were the interval is set to 10 and at 3 seconds when the interval was set to 90. These failures result from the increased overhead caused by either the frequent invocation of the Task Scheduler (interval 10) or from the delayed invocation of the Task Scheduler (interval 90) resulting in a significant increased number of essentially critical task instances. The slightly better survivability performance when the interval was set to 90 is a result of the delayed invocation of the Task Scheduler making it easier for essentially critical task instance to acquire their need resource since there are in fact fewer task instances (as was seen in the middle criticality experiments). It can be clearly seen from these charts (and in fact all charts) that the Periodic model, since its survivability performance was so poor over the entire range of periodic intervals, was unable satisfy the stringent requirement of survivability (that no essentially critical task instance should fail).

As was shown in both the low criticality and middle criticality experiments, the performance of the Dynamic model is not effected by the interval chosen, since it does not utilize this parameter at all in its decision to invoke the Task Scheduler. Its performance matched those characteristics observed under low and middle criticality, with only slight changes in deadline failure rate due to increased number of essentially critical task instances. However, in none of the charts did this slightly increased deadline failure rate change the relative position of the Dynamic model in respect to the Periodic model and Adjusted model. Due to the extremely high level of criticality and therefore the increased number

of essentially critical task instances this model is no longer able to complete the entire experimental run. However, its performance in respect to survivability is in all charts at least as good if not superior to the best performing model. In no chart does its survivability performance match the relatively poor performance of the Periodic model. It was very clear that in respect to survivability the Dynamic model clearly was able to adjust its invocation of the Task Scheduler to meet the changing environmental conditions, therefore allowing it to survive for significantly longer periods of time.

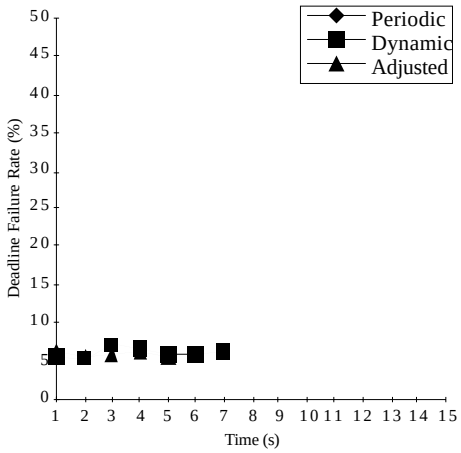


Figure 9-21: High Criticality (Period=10, FTh=1, STh1=1, STh2=1)

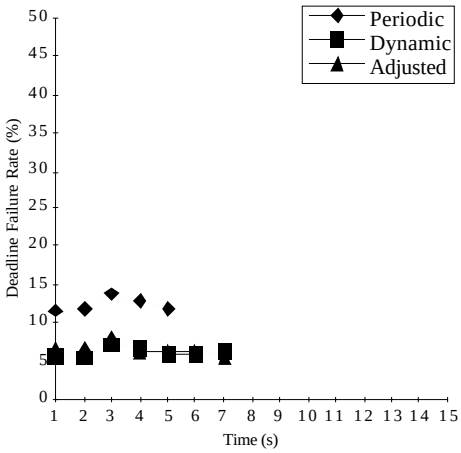


Figure 9-22: High Criticality (Period=50, FTh=1, STh1=1, STh2=1)

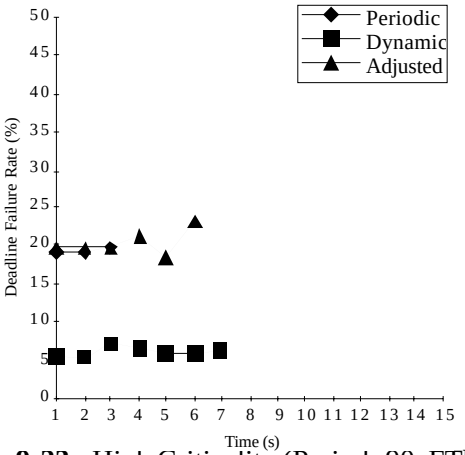


Figure 9-23: High Criticality (Period=90, FTh=1, STh1=1, STh2=1)

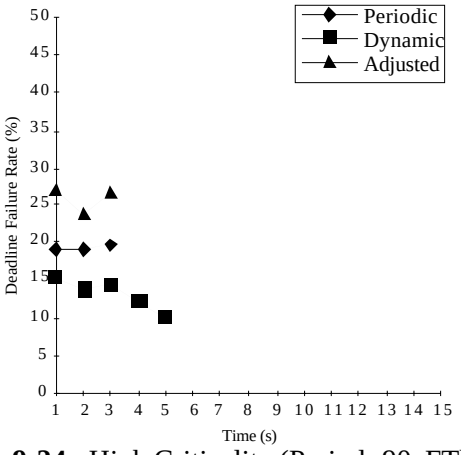


Figure 9-24: High Criticality (Period=90, FTh=1, STh1=1, STh2=5)

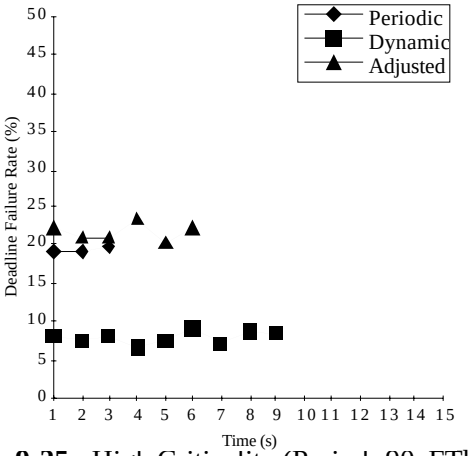


Figure 9-25: High Criticality (Period=90, FTh=3, STh1=1, STh2=5)

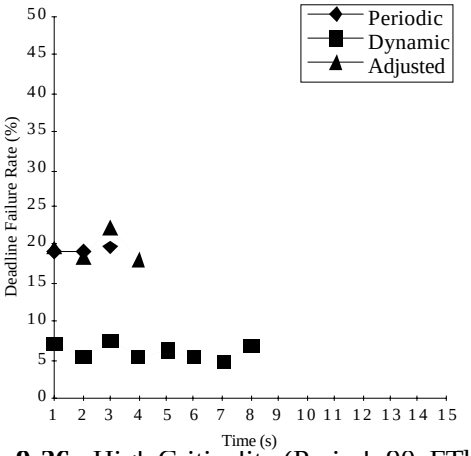


Figure 9-26: High Criticality (Period=90, FTh=5, STh1=1, STh2=5)

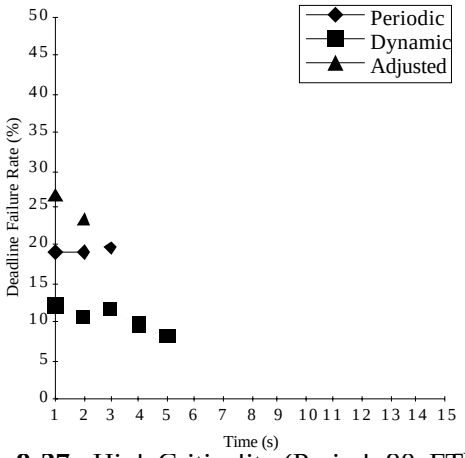


Figure 9-27: High Criticality (Period=90, FTh=5, STh1=3, STh2=5)

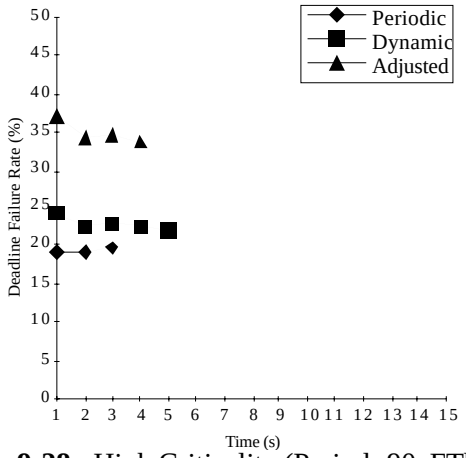


Figure 9-28: High Criticality (Period=90, FTh=5, STh1=5, STh2=5)

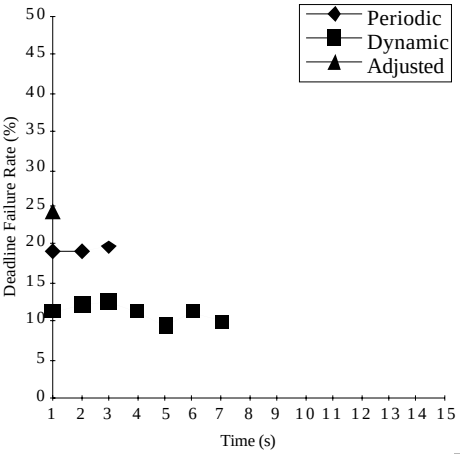


Figure 9-29: High Criticality (Period=90, FTh=5, STh1=3, STh2=3)

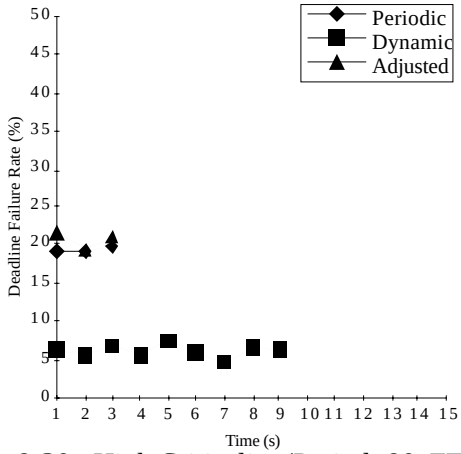


Figure 9-30: High Criticality (Period=90, FTh=5, STh1=1, STh2=1)

The performance of the Adjusted model in respect to deadline failure rate clearly showed the same tendencies as in the low and middle criticality experiments. In respect to the survivability of the model, the Adjusted model was able to survive for at least as long as (if not longer than) the Periodic model in all charts. However, its survivability performance never exceeded that of the Dynamic model (only matching the performance under certain parameter settings). More important is its characteristic tendency to match the performance of the Periodic model where the period is set relatively high (period set to 90). Here the Adjusted model's survivability performance is drastically effected by the poor interval setting and its survivability reflects this since it's unable to survive for more than 4 seconds.

In these experiments it was clearly shown that the Periodic model would be unable to handle safety-critical environments were not only is the deadline failure rate performance important, but more important is the survivability performance. Only integration models that are able to adjust to the changing environmental conditions (Dynamic and Adjusted models) were able to handle to a significantly greater degree the more important survivability measurement. Only under extreme settings of the criticality of the task profile (as seen in these experiments) did the Dynamic model not complete the entire experimental run. While the Adjusted model was able to match the survivability performance of the Dynamic model (under certain settings), it was never able improve on the performance of this model. In conclusion it's very clear that for systems working in safety-critical environments where not only is the deadline failure rate performance important but more important is the survivability performance that the Dynamic model (in particular through its simplicity) clearly outperformed both the Periodic and Adjusted models. It was therefore choosen as the Task Scheduler/ File Server integration policy implemented in MELODY.

Chapter 10 File Assigner Integration Experiments

Remember that in the MELODY model a trade-off has to be made, under changing request and deadline failure patterns, between the costs of serving file requests with a given distribution of public and private copies, and the costs for realizing various alternative distributions. The term cost here denotes time delays for overhead operations, or for communication and transmission delays, both for remote and local communication. A large number of public copies is advantageous for sensitive read tasks since such a copy is then more likely to be locally available. However, write tasks at the same time suffer from this since the cost for updating grows with the number of copies. The alternatives for changing the configuration of public copies are:

- relocating a public copy to the requesting site.
- making an additional public copy available to the site of the requesting task.
- deleting a file copy if there were not enough requests over a period of time.

Each local File Assigner cooperates with the File Server or Task Scheduler, and with remote File Assigners in order to manage replication, relocation, and deletion of files within the MELODY file system. More details regarding the actions of the File Assigner can be found in chapter 6.

For integrating the File Assigner into the other MELODY functions two different models had been described in section 6.1.4: the Task Scheduler-Oriented Integration (*TS-Oriented*) and the File Server-Oriented Integration (*FS-Oriented*). While in the first model FA is invoked early on behalf of task instances, the information about task abortion in the second model is more accurate though each task instance is treated comparatively late in its life cycle.

Let us also recall that the integration of the File Assigner (client and server) will be controlled by the File Server in both models. In particular, file access requests are interleaved with the FA actions in accordance with the operation priority. All requests are separated into two distinct categories, *essentially critical requests* (on behalf of essentially critical task), and *non-essential requests*. Both the request types are further separated into *current access requests* and *future assignment requests*. Future assignment requests contain all local or remote File Assigner requests and responses for the relocation, replication and deletion of shared files. The File Server will process essentially critical request or non-essential requests until it has to stop under the rules of the dynamic model (see chapter 9). The interdependencies of the TS-Oriented model can be seen in figure 10-1, while the interdependencies of the FS-Oriented model can be seen in figure 10-2.

To study the trade-off between these two integration models a large number of distributed experiments were performed. In the experiments discussed here, 100 task were distributed amongst all the sites. 16 data files were distributed with each file initially having one public copy and no private copies. Parameter ranges for the task profiles (from which initial settings for the tasks in the profile were made) are shown in table 10-1. In the following task profiles, task deadlines had then been modeled accordingly to fit tightly. The rather high deadline failure rates (which can be seen in the following figures) were thus to be expected. They are not unacceptable as long as survivability is guaranteed, i.e. the failed the failed task instances had not yet been essentially critical. In this way characteristic differences in the performance of the models under a relatively high load could be more accurately determined.

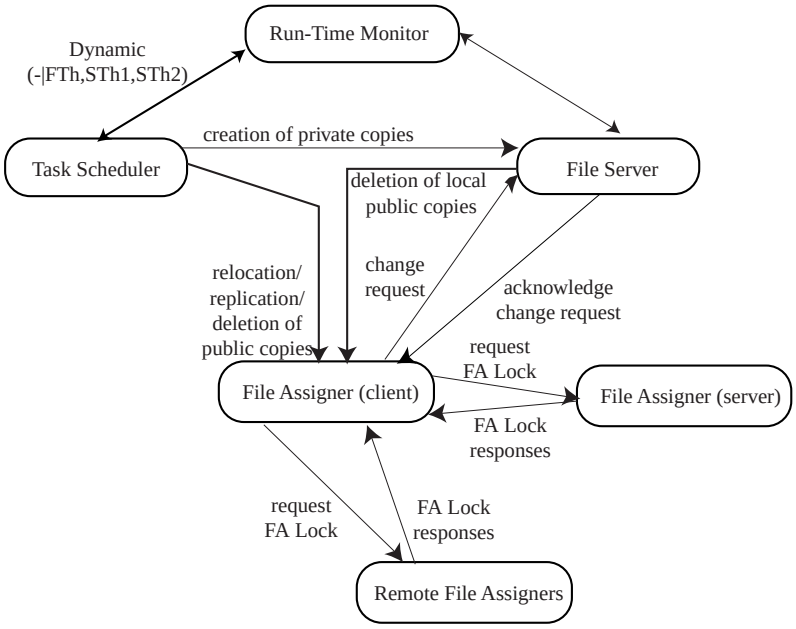


Figure 10-1: File Assigner: Task Scheduler Oriented Integration Model

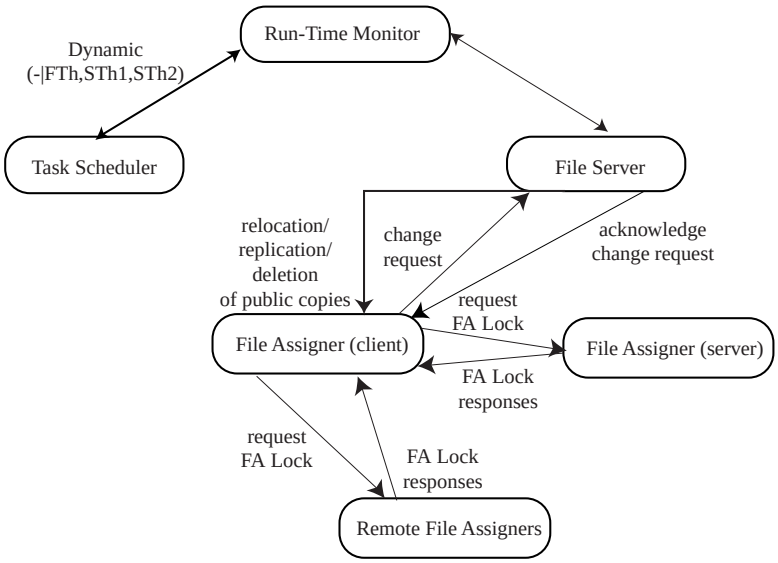


Figure 10-2: File Assigner: File Server Oriented Integration Model

Task Parameter	Range of Values	
	Read Access	Write Access
Deadline	12..15 msec	70..95 msec
Worst Case Execution Time	5 msec	10 msec
Next Arrival Time	230..330 msec	350..450 msec
Criticality (C_j)	varied [5..11], [7..13], [9..15]	
Sensitivity (R_j)	varied [6..9], [9..12], [12..15]	
# of Required Files	2-4	
Read / Write Dominance	Write Dominance: 25% readers / 75% writers	

Table 10-1: File Assigner Integration Experiments Task Profile Parameters

The experiments were performed on a Token Ring of 7 IBM RS/6000 machines. Here MELODY is implemented by using the AIX kernel functions. Its commands are assigned the highest possible (absolute) priority in AIX. The ranges for criticality and sensitivity values were both [0,15], with the thresholds set as a_i'=b_i':=2; a_i"=b_i":=8. The experimental parameters chosen showed the tendencies between models very clearly. The uniformity over all experimental set-ups was chosen because of

representational simplicity not because different parameter ranges exhibited different tendencies. The **performance measurements** (as in prior experiments) were based on the percentage of task failures during a given time interval (1 second), and equally on **survivability**. *Each experiment was run 10 times with results averaged*. Beyond this we did up to 30 runs for better judgment on survivability but found no significant differences compared to the smaller number of runs which is then the basis of the report. In addition to this we measured survivability in the following way: If an essentially critical task failed during one of the experimental runs then all other runs were counted as failed at the same failure time at which the one run had experienced the failure.

Using the above task profile parameters the following three distinct task profiles were developed to test the models under varying ranges of read/write dominance:

Read Dominance: In this task profile there was a dominance (75%) of reading tasks,
Even Mix: This profile contained an even distribution of read and write tasks (50%/50%),
Write Dominance: In this profile there was a significant dominance (75%) of writing tasks.

In each experiment the TS-Oriented integration model was compared to the FS-Oriented integration model as well as against a version of MELODY with File Assigner disabled. The disabled model would not allow for any changes in the distribution of file copies. The model then has the advantage of a very low overhead and acts as a ***Base-line*** model. Criticality for each of the profiles was varied between the ranges of [5..11], [7..13] and [9..15]. Sensitivity for each of the profiles was varied between the ranges of [6..9], [9..12] and [12..15]. This resulted in 9 distinct sets of experiments for each of the three task profiles, and in this manner the results clearly showed the influence criticality, sensitivity and read/write dominance on the deadline failure rate performance and survivability performance of the three integration models (TS-Oriented, FS-Oriented and Base-line).

10.1 Read Dominance

In an environment where the dominance of read operations is high, the urgency to have a local file copy increases due to tendency of read operation to require fast access to file copies. In this situation both the TS-Oriented and FS-Oriented integration models should improve the Base-line model's performance (in respect to the deadline failure rate) due to their invocation of the File Assigner. It is expected that the deadline failure rate performance of the FS-Oriented model should be significantly better than that of the TS-Oriented integration model, due to the more accurate information regarding the access requirements of task instances at a certain site. It is expected that the survivability performance of the TS-Oriented integration model should be better than the FS-Oriented integration model. This being due to its earlier invocation of the File Assigner and as a result increased chances that a copy would be available before the next invocation of a nearly essentially critical task.

All figures clearly show a significantly poor start for both integration models (TS-Oriented and FS-Oriented) compared to the Base-line integration model demonstrating the heavy overhead load (resulting from relocation and replication of public file copies) of the more sophisticated models. In the task profiles with the sensitivity set in the range of [6..9] (figure 10-3) this overhead was the result of public copy relocation and replication since the majority of task instances could only utilize information from public copies. When the range of sensitivity was between [12..15] (figure 10-5) the overhead was found to result from the creation of private copies. This overhead was caused by both public copy relocation and replication and private copy creation when the sensitivity was set between the range of [9..12] (figure 10-4). After only 3 to 4 seconds the integration models are performing distinctively superior to the Base-line integration model (as can be seen in figure 10-3 and all Read Dominance charts). For the criticality range [9..15] (nearly all non-critical tasks) there is a clear deadline failure rate performance increase by the FS-Oriented integration model over the TS-Oriented integration model. This is a direct result from the more accurate decision made by the FS-Oriented integration model in respect to improving the performance of access to file copies (figure 10-3, figure 10-4 and figure 10-5). No significant performance change can be seen as the sensitivity of the task profile is varied from [6..9] to [12..15]. This is due to the fact that under such a high read dominance the additional overhead resulting from public copies is not significantly more than for private copies.

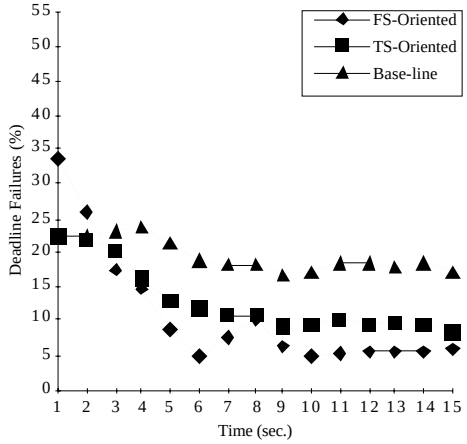


Figure 10-3: Read Dominance (C_j[9..15] / R_j[6..9])

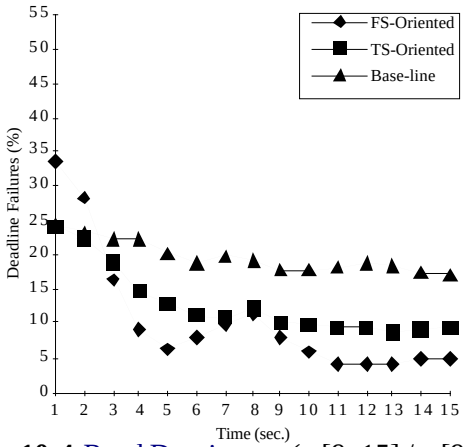


Figure 10-4: Read Dominance (C_j[9..15] / R_j[9..12])

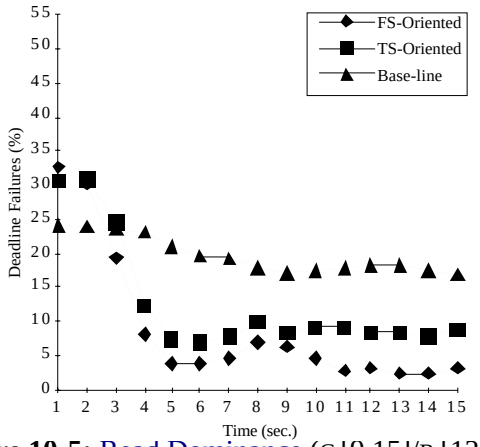


Figure 10-5: Read Dominance (C_j]9.15]/R_j[12.15])

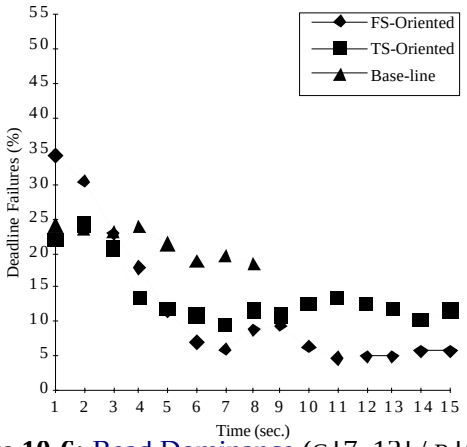


Figure 10-6: Read Dominance (C_j[7..13] / R_j[6..9])

As the criticality of the task profile is increased (range 7 to 13) clearer tendencies regarding the survivability of the models become apparent. In a middle range of criticality (figure 10-6, figure 10-7 and figure 10-8) the characteristic tendency in regards to both integration model's deadline failure rate performance improvement is still quite apparent, with the FS-Oriented integration model improving on the performance of the TS-Oriented integration model. The increased criticality causes significant problems for the Base-line integration model as it is no longer to complete the entire experiment (failing between the 8th and 9th second in figure 10-6). Both integration models are able to survive the entire experimental run.

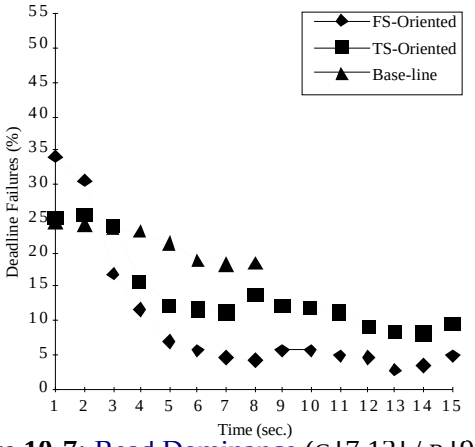


Figure 10-7: Read Dominance (C_j[7.13] / R_j[9.12])

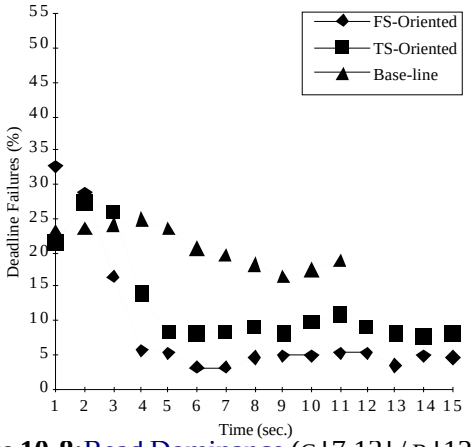


Figure 10-8: Read Dominance (C_j[7.13] / R_j[12.15])

A significant increase in the criticality of the task profile (range 5 to 11) caused even more problems for the Base-line integration model. Under this high level of criticality, the Base-line

integration model is only able to survive up to the 5th second under all ranges of sensitivity (figure 10-9, figure 10-10 and figure 10-11). The characteristic tendency in regards to both integration model's deadline failure rate performance improvement is still quite apparent, with the FS-Oriented integration model improving on the performance of the TS-Oriented integration model. Both integration models however are no longer able to survive the entire experimental run and fail between 7 and 10 seconds. The TS-Oriented integration model does show a performance increase over the FS-Oriented integration model in respect to survivability (figure 10-9 and figure 10-10) as it is able to survive between 9 and 10 seconds, while the FS-Oriented integration model is only able to survive only for 7 seconds. This performance increase results from the TS-Oriented integration model's early invocation of the File Assigner. This early invocation of the File Assigner increases the chances that a private copy would be available before the arrival of an essentially critical task instance. When the sensitivity of the task profile is increased between the range of 12 to 15 (figure 10-11), no survivability performance increase can be seen between the two integration models, since the FS-Oriented integration model is creating additional private copies (due to the high dominance of robust tasks) and is able to survive for 9 seconds (matching the survivability performance of the TS-Oriented integration model).

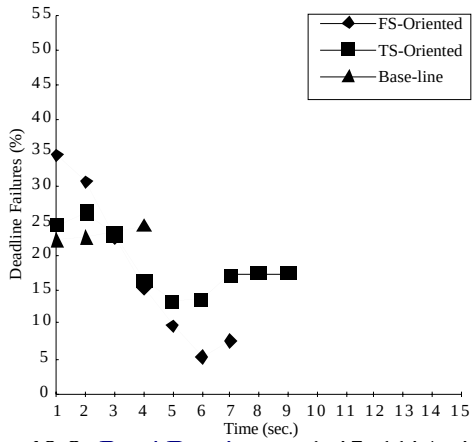


Figure 10-9: Read Dominance (C_j[5..11] / R_j[6..9])

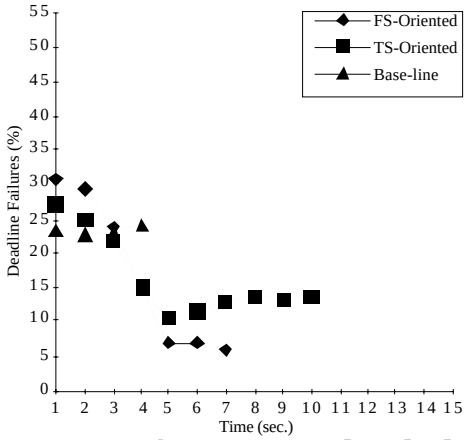


Figure 10-10: Read Dominance (C_j[5.11]/R_j[9..12])

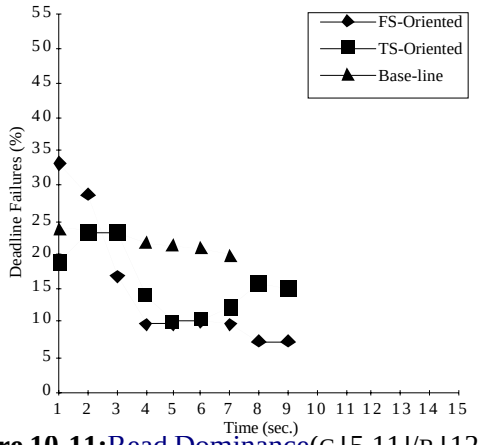


Figure 10-11: Read Dominance(C_j[5.11]/R_j[12.15])

10.2 Even Mix

It is expected, as in the Read Dominance experiments, that both the TS-Oriented and FS-Oriented integration models should improve the performance of the Base-line integration model in respect to the deadline failure rate performance due to their invocation of the File Assigner. It is also expected that the deadline failure rate performance of the FS-Oriented model should be significantly better than that of the TS-Oriented integration model (as in the Read Dominance experiments). The survivability performance of the TS-Oriented integration model should still outperform both the FS-Oriented integration and Base-line models.

All figures clearly showed a significantly poor start for both integration models (TS-Oriented and

FS-Oriented) compared to the Base-line integration model (figure 10-12, figure 10-13 and figure 10-14), as was observable in the previous Read Dominance experiments. After only 3 to 4 seconds, their deadline failure rate performance had also improved and was clearly better than the Base-line model (as in the previous Read Dominance experiments). The characteristic tendency of the FS-Oriented integration model to outperform the TS-Oriented integration in respect to deadline failure rate performance was also observable in all figures.

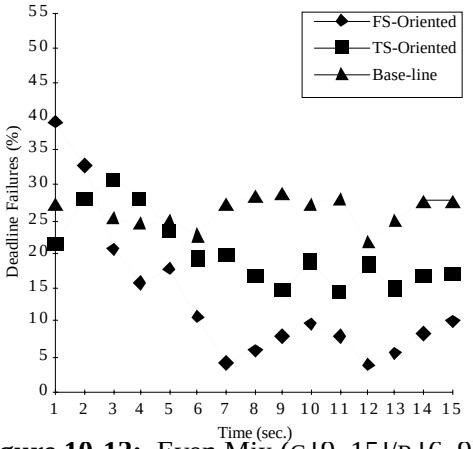


Figure 10-12: Even Mix (C_j[9..15]/R_j[6..9])

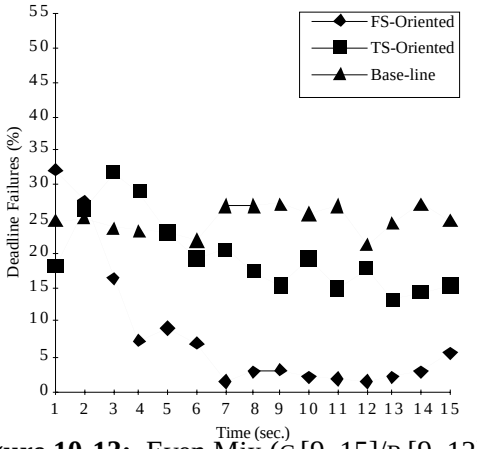


Figure 10-13: Even Mix (C_j[9..15]/R_j[9..12])

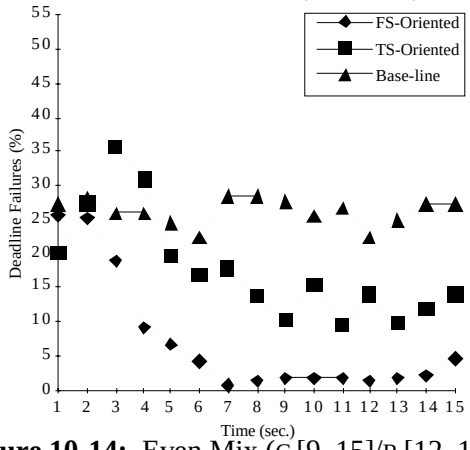


Figure 10-14: Even Mix (C_j[9..15]/R_j[12..15])

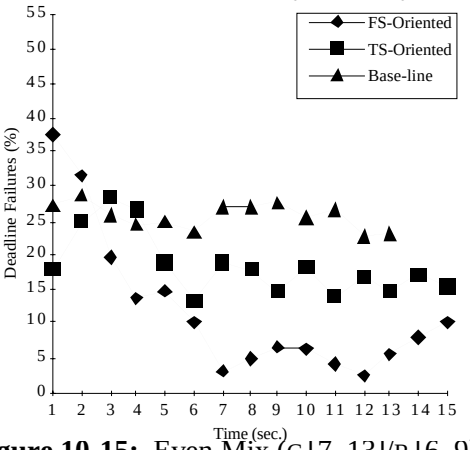


Figure 10-15: Even Mix (C_j[7..13]/R_j[6..9])

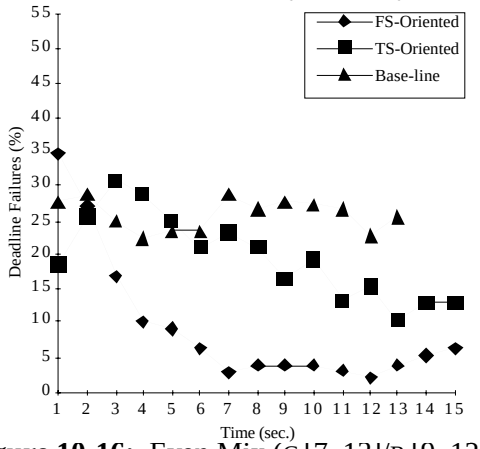


Figure 10-16: Even Mix (C_j[7..13]/R_j[9..12])

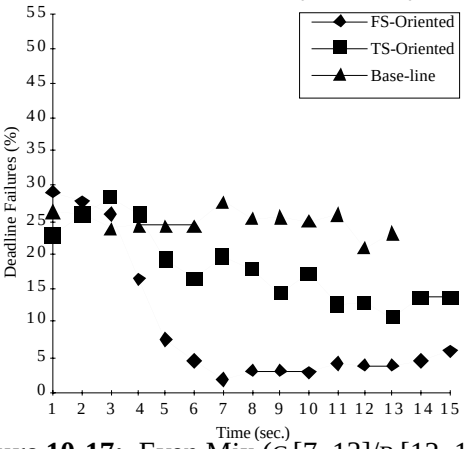


Figure 10-17: Even Mix (C_j[7..13]/R_j[12..15])

As the level of criticality is increased (range 7 to 13), the characteristic tendency of the Base-line model to fail (here after the 13th second), under increasing levels of criticality, can be seen in figure 10-15, figure 10-16 and figure 10-17. Both the FS-Oriented integration model and the TS-Oriented integration model were again able to survive the entire experimental run. Their relative performance matched that seen under the low level of criticality. Otherwise no significant performance differences can be seen between the models.

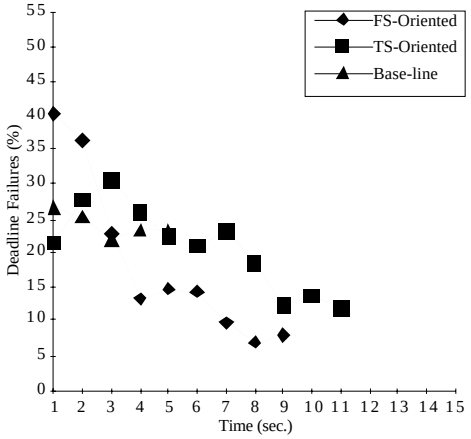


Figure 10-18: Even Mix (C_j[5..11]/R_j[6..9])

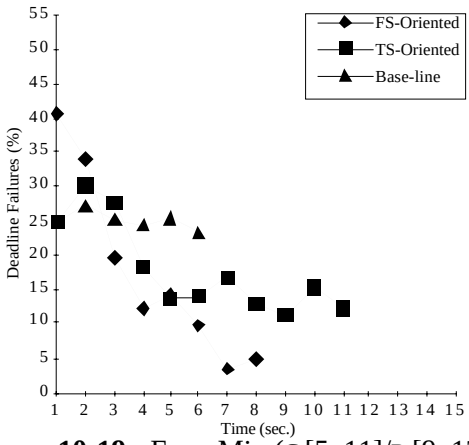


Figure 10-19: Even Mix (C_j[5..11]/R_j[9..12])

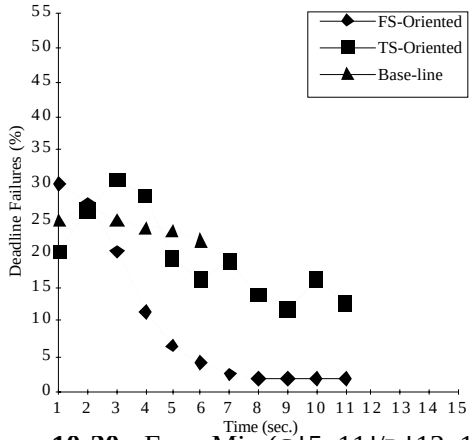
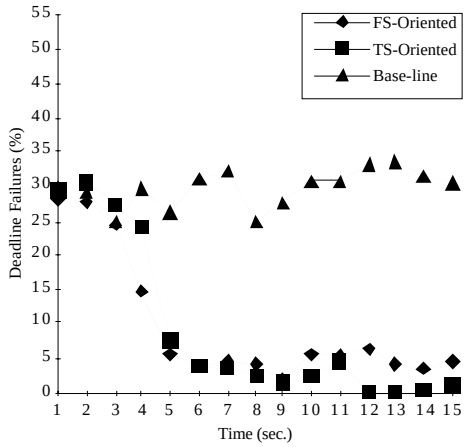


Figure 10-20: Even Mix (C_j[5..11]/R_j[12..15])

Further increasing the level of criticality (range 5 to 11), showed that the characteristic tendency of the Base-line model to fail (here between 5 and 6 seconds) could also be seen in figure 10-15, figure 10-16 and figure 10-17. Again under higher levels of criticality both the FS-Oriented integration model and the TS-Oriented integration model failed between 9 and 11 seconds. The TS-Oriented integration model still shows a significant performance increase over the FS-Oriented integration model in respect to survivability (figure 10-18 and figure 10-19) as it is able to survive for 11 seconds, while the FS-Oriented integration model is only able to survive between 8 and 9 seconds. This performance increase results from the TS-Oriented integration model's early invocation of the File Assigner. This early invocation of the File Assigner increases the chances that a private copy would be available before the arrival of an essentially critical task instance. When the sensitivity of the task profile is increased between the range of 12 to 15 (figure 10-20), no survivability performance increase can be seen between the two integration models, since the FS-Oriented integration model is creating additional private copies (due to the high dominance of robust tasks) and is able to survive for 9 seconds (matching the survivability performance of the TS-Oriented integration model).

10.3 Write Dominance

As it was expected in previous experiments, both the TS-Oriented and FS-Oriented integration models should improve the performance of the Base-line integration model in respect to the deadline failure rate performance due to their invocation of the File Assigner. It is also expected that the deadline failure rate performance of the FS-Oriented model should be better than that of the TS-Oriented integration model (as in previous experiments). The survivability performance of the TS-Oriented integration model should be better than both the FS-Oriented integration model and Base-line model. The improved survivability performance resulting from the earlier invocation of FA through a Task Scheduler decision to not schedule a number of task instances.



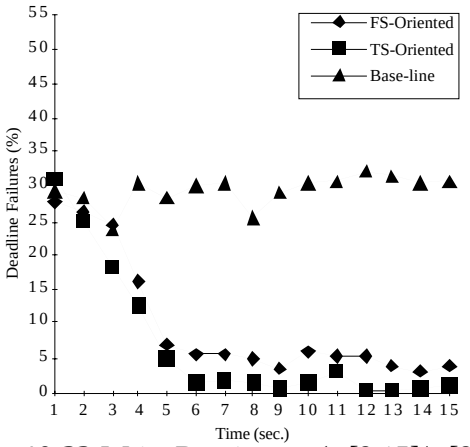
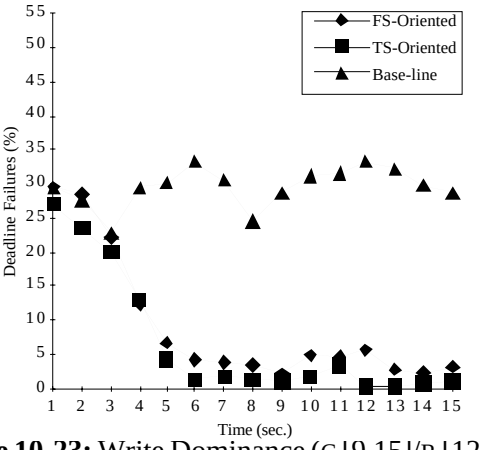


Figure 10-21: Write Dominance (C_j[9..15]/R_j[6..9]) **Figure 10-22:** Write Dominance (C_j[9.15]/R_j[9.12])

The characteristic poor deadline failure rate performance during the first few seconds of an experiment for both integration models (TS-Oriented and FS-Oriented) in comparison to the Base-line integration model is no longer observable (figure 10-12, figure 10-13 and figure 10-14). This results from the fact that the overhead to delete a public copy is not high enough to cause serious deadline failure problems for other tasks. For task profiles were there is a dominance of write operations the benefit of deleting public copies is so significant that the small communication overhead is nearly negligible. Therefore, in all charts the deadline failure rate performance of both the FS-Oriented integration model and TS-Oriented integration model improves significantly after 4 to 5 seconds.



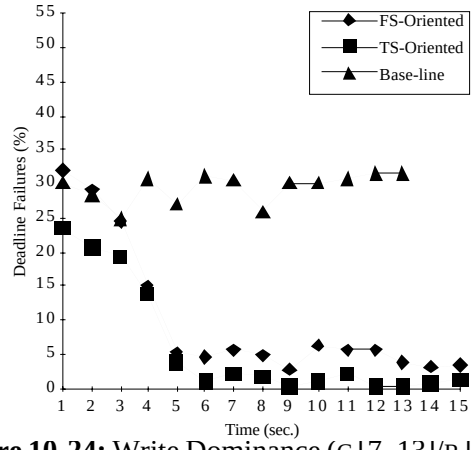
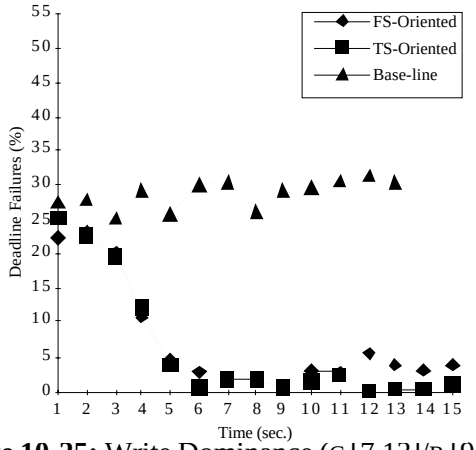


Figure 10-23: Write Dominance (C_j[9.15]/R_j[12.15]) **Figure 10-24:** Write Dominance (C_j[7..13]/R_j[6..9])



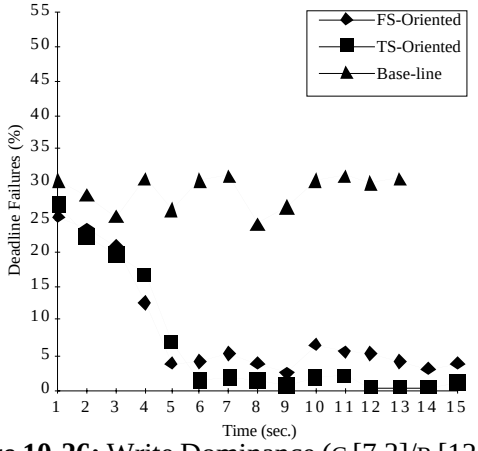
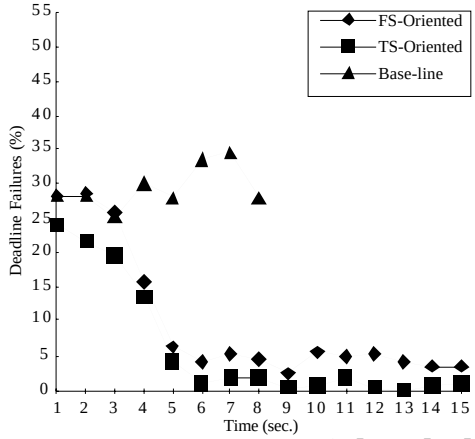


Figure 10-25: Write Dominance (C_j[7.13]/R_j[9.12]) **Figure 10-26:** Write Dominance (C_j[7.3]/R_j[12.15])

As the level of criticality is increased (range 7 to 13), the characteristic tendency of the Base-line model to fail (here after the 13th second) under increasing levels of criticality can be seen in figure 10-

24, figure 10-25 and figure 10-26. Both the FS-Oriented integration model and the TS-Oriented integration model were again able to survive the entire experimental run. Their relative performance matched that seen under the lower level of criticality. Otherwise no significant performance differences can be seen between the models.



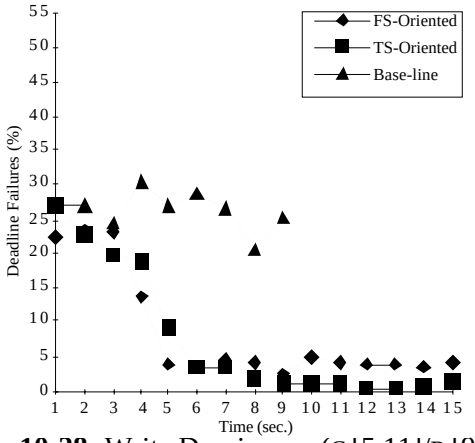


Figure 10-27: Write Dominance (C_j[5..11]/R_j[6..9])**Figure 10-28:** Write Dominance (C_j[5.11]/R_j[9.12])

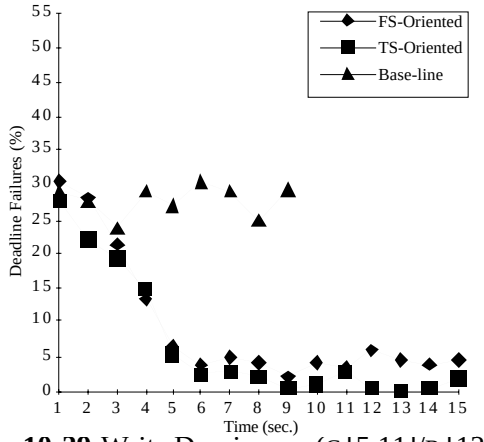


Figure 10-29: Write Dominance (C_j[5.11]/R_j[12.15])

Again the characteristic tendency of the Base-line model to fail (here between 5 and 6 seconds), as the level of criticality is increased (range 5 to 11), could also be seen in figure 10-27, figure 10-28 and figure 10-29. However, in contrast to previous experimental results, under high level of criticality both the FS-Oriented integration model and the TS-Oriented integration model completed the entire experimental run. The TS-Oriented integration model no longer shows a significant survivability performance increase over the FS-Oriented integration model. The deadline failure rate performance of both models is nearly identical. This significant change in both the deadline failure rate performance and survivability performance is a direct result from the relative ease (lower overhead) to delete a public copy. Therefore, when a nearly essentially critical task instance is detected the small time variation in invoking FA by the two models no longer has a significant impact on whether or not the models survive, since both models are able to remove public copies before the next arrival of the essentially critical task instance.

In conclusion, in all experiments only after a few seconds the sophisticated integration models performed distinctively superior to the less sophisticated Base-line model. For profiles with a large number of read operations the deadline failure rate performance of the FS-Oriented integration model was clearly superior to that of the TS-Oriented integration model. Only under a high dominance of write operations did the deadline failure rate performance of the TS-Oriented integration model match that of the FS-Oriented integration model. For the criticality range [7..13] the Base-line integration model fails very early (between 5 and 9 seconds), in contrast to the integration models which are able to survive the entire experimental run. As the criticality is increased (range [5..11]), the integrated File

Assigner models do fail. However, in all charts the survivability performance of the Base-line model is always much worse than either the FS-Oriented integration model or the TS-Oriented integration model. It was also shown that under a high dominance of write tasks that the advantage of the TS-Oriented to invoke FA early than the FS-Oriented model was no longer significant enough to show a difference in the survivability performance. This change in the survivability performance resulting from the relative ease (low overhead) to delete a public copy offsetting the difference between the invocation times of the two models. Given the wide range of read/write variance, criticality and sensitivity, the FS-Oriented integration model clearly outperformed the TS-Oriented integration model in terms of deadline failure rate performance, while matching the performance in terms of survivability, therefore the FS-Oriented integration model was chosen as the File Assigner integration model implemented in MELODY.

Chapter 11 Task Monitoring Integration Experiments

The task monitor monitors all task instances currently competing for access to shared files (local/ remote). A brief overview of the task monitor is included here to clarify the development of the integration models used (more details can be found in chapter 7). Monitoring is done by determining an estimate of the remaining time required to acquire all needed resources and complete the task instance's computation phase prior to its associated deadline. If the task monitor determines that a task instance can no longer complete before its deadline, then it will be abort. The following estimates are used to determine the remaining time required by T_{jk}:

Estimated Acquisition Time (EAT_{jk}): This is the estimated time required by task instance T_{jk} to acquire locks for all needed resources. For a reading T_{jk} this is the time from which the request to obtain a read-lock is sent, to the site holding a copy of the file required, until the read-lock is returned to T_{jk}. Once a shadow or local private copy has been acquired by T_{jk} it is guaranteed (by the remote FS) not to be physically deleted until either T_{jk} has completed its execution phase or has released the resource itself. For a writing T_{jk} this estimates the time interval from the point at which the access request is broadcast, to all sites holding a public copy, until all write-locks have been received by T_{jk}. This includes both phases of the Delayed Insertion protocol (see section 4.1.1).

Estimate Locking Time (ELT_{jk}): For a writing task instance T_{jk} the task monitor also determines an estimate for the time required to obtain all write-locks once all *ready* messages have been received by the task instance (once phase 1 of the Delayed Insertion protocol has been completed). This estimate includes the time from which the schedule request message is broadcast, to all sites holding a public copy of the file, until all write-locks have been received by T_{jk}.

Estimated Computation Time (ECT_{jk}): This is an estimate for the time required to complete the read/write operation requested by the task instance T_{jk}. For reading task instances this is the time from which the read request is sent, to the site holding the copy (shadow or private) that is to be read, to the point at which the result of the read operation is received by T_{jk} at the local site. For writing T_{jk}s this is the time from which the write operation is broadcast, to all sites holding a copy of the file, to the point at which the operation has been completed at those sites.

In the current distributed implementation of MELODY the estimates used to determine any of the time values above is derived from the last ten instances of task T_j.

The determination as to whether or not a reading task instance T_{jk} has completed before its associated deadline, is made when the result of the read operation is received by T_{jk}. If the message is received prior to deadline DT_{jk} then T_{jk} is determined to be successful. Otherwise, T_{jk} has failed and the relative degree of criticality and sensitivity of the next task instance of T_j will reflect this. Any message received after the deadline would be ignored, since the information is considered to be obsolete by T_{jk}. However, the FS guarantees that once a writing task instance T_{jk} has broadcasted its update operation to all sites (involved in the write operation) the update will be completed regardless of whether the operation would be completed prior to deadline DT_{jk}. This ensures consistency of the MELODY file copies since some sites may have already performed the update operation prior to the deadline expiring (MELODY also provides no mechanism to handle a roll-back in order to avoid the high overhead this would create). As a result, the determination of whether or not a writing task instance T_{jk} is successful, is made based on the completion status of the write operation returned by each of the sites performing the update operation. If any site returns a "failed" status (meaning the operation completed after deadline DT_{jk}) then T_{jk} is marked as having failed.

Once estimates for T_{jk} have been determined, the task monitor sets a number of sub-deadlines

corresponding to phases during the task execution life cycle (see figure 3-4) based on the deadline DT_{jk} (determined by the application). The following sub-deadlines allow the task monitor to determine whether to abort T_{jk} based on the phase T_{jk} is in and whether the corresponding phase sub-deadline has expired:

Location Sub-Deadline (DLo_{jk}): This sub-deadline is set based on task instance T_{jk}'s ECT_{jk} and EAT_{jk}. It is used to determine if the task instance's location phase has finished in time to complete the remaining acquisition and computation phases. The deadline DLo_{jk} is determined by subtracting ECT_{jk} and EAT_{jk} from the value of DT_{jk}.

Allocation Sub-Deadline (DAl_{jk}): This sub-deadline is set based on task instance T_{jk}'s ECT_{jk} and ELT_{jk}. It is used to determine if the writing task instance has completed phase 1 of the Delayed Insertion Protocol in time to finish Phase 2 and the task instance's computation phase. The deadline DAl_{jk} is determined by subtracting ECT_{jk} and ELT_{jk} from the value of DT_{jk}. This sub-deadline is only set for writing task instances, since a reading task instance is not required to obtain ready messages from any the files in it's list of required files (LRF_{jk}).

Acquisition Sub-Deadline (DAc_{jk}): This sub-deadline is set based only on task instance T_{jk}'s ECT_{jk}. It's only used to determine if T_{jk} has obtained all necessary locks in time to finish its computation phase. The deadline DAc_{jk} is determined by subtracting ECT_{jk} from the value of DT_{jk}.

A more detailed description of the policies used by the task monitor can be found in section 7.1.2.

Using minimum values for the estimates to abort a task instance as late as possible creates a very weak requirement to begin the task instance's next phase. However, it more accurately aborts only those task instances that have almost no chance to complete before their deadline. This optimistic view of a task instance's ability to complete before its deadline causes many task instances to begin phases (which creates additional overhead from message communication and possibly unnecessary locking of copies) where they have no chance to complete (the time required is greater than the minimum). Using maximum values for the estimates creates a very strong requirement (aborting a task instance as early as possible) to begin the task instance's next phase. This pessimistic view of the task instance's ability to complete before its deadline causes many task instances to be aborted that have a fair (or reasonable) chance to meet their deadline. A large number of distributed experiments were conducted in order to study the trade-offs between these optimistic and pessimistic perspectives.

Experiments were performed on a Token Ring of 7 IBM RS/6000 machines. Here MELODY is implemented by using the AIX kernel functions. Its commands are assigned the highest possible priority in AIX. For every experiment 10 runs were performed. Beyond this we did up to 30 runs for better judgment on survivability but found no significant differences compared to the smaller number of runs which is then the basis of the report. In each of them the criticality and sensitivity values were both in the range of [1..15]. The thresholds for criticality and sensitivity were set to $a_i' = b_i' := 2$; $a_i'' = b_i'' := 8$. Other information about the task profiles can be found in their associated sections. Performance was measured as the total number of deadline failures during a 2 second interval of time. If an essentially critical deadline was missed during an experimental run, the remaining execution of all 10 runs after the point at which the essentially critical task failed, was disregarded. In this way combined insights into both the different real-time behavior perspectives of deadline failure rate and survivability could be derived. Recall that a system is called survivable if and only if every essentially critical task instance meets its deadline [Dan92 and WeL97].

In all experiments a base model (denoted as ***Base***) was developed in which a **task instance is only aborted if the deadline DT_{jk} has expired**. This model is used to show the performance of the MELODY system without any monitoring of task instances during their acquisition phase. This model would then have the most optimistic view of a task instances chances to complete prior to its deadline. This results in the lowest amount of additional overhead caused by task monitoring, while resulting in the largest amount of overhead caused by competing task instances (that relatively have no chance to complete prior to their deadline). This model could then be used to evaluate the benefits of early abortion of task instances in comparison to the additional overhead caused by task monitoring. To abort task instances earlier than the deadline the following model was developed which tries to ensure that no task instance

begins its competition phase that does not have a fair chance to complete prior to its deadline.

Abort before Computation (AbC): In this model a task instance would be aborted if the sub-deadline DAc_{jk} had expired. The execution time of a read/write operation at a site is assumed to be known (or within a well defined range). The message delay for sending and receiving messages has been found to be largely constant (within a quite narrow tolerance range close to the minimal values, and this range is only very rarely surpassed). Therefore, in all experiments average values for the estimated computation time (ECT_{jk}) were used.

In the following three models a task instance would not only be aborted based on the sub-deadline DAc_{jk} (as in the AbC model) but also during a task instance's acquisition phase based on the two sub-deadlines DLo_{jk} and DAl_{jk}. These models try to ensure that only those task instances that have a chance to allocate and lock there required files, are allowed to compete for access to shared resources. In order to cope with the large variation in the time required to complete a task instance's acquisition phase (which varies due to the level of competition) the following three models were developed (based on the estimate used for EAT_{jk} and ELT_{jk}):

Earliest Abort before Acquisition (EAbA): This is the most pessimistic model using maximum values for estimates of EAT_{jk} and ELT_{jk}. This model tries to abort a task instance at the earliest point of time during its acquisition phase, therefore reducing the level of competition in the system as much as possible.

Medium Time Abort before Acquisition (MAbA): Using average values for both EAT_{jk} and ELT_{jk} allows this model to try and adapt to changing competition levels. Therefore, abort only those task instances that do not have a reasonable chance to complete.

Latest Abort before Acquisition (LAbA): Using minimum values for estimates of EAT_{jk} and ELT_{jk} gives this model a very optimistic view of a task instance's chances to complete before its deadline. This ensures that a task instance is aborted at the latest point of time, and only if it has almost no chance to complete before it's deadline.

Again, as mentioned previously, the maximum, minimum and average values refer to the values of the prior 10 task instances of task T_j.

11.1 Deadline Failure Rate Performance

As the level of competition in the system increases, the benefit of aborting task instances as soon as possible increases, (this in effect reduces the competition at remote resources and reduces the overhead from unnecessary locking and execution against remote resources). The first set of experiments varies the number of writing tasks at a site between 1 and 12 that compete for access to the same shared file copies. Each task instance updates 256 records (20K of data) in one file containing 1024 records (80K or data). The deadline was set to 40ms after the creation time of the task instance. The next arrival of a task instance was then determined to vary between 15ms and 25ms after the deadline of the prior task instance. These settings for the deadline and next arrival times for the tasks created a range of environments were the level of competition for a file copy varied from a initially very low level of competition (1 task at each site) to a very high level of competition (12 tasks at each site). All task instances in these experiments are non-critical and essentially sensitive (for writing tasks sensitivity is irrelevant since any update operation must be performed against all public copies and no decision is made whether to use a private copy). This keeps the File Assigners inactive thus allowing us to study the tendencies of the task monitoring models (regarding solely the deadline failure rate performance), without interference of overhead effects brought about by the FA (creation and deletion of public or private copies at a site). Any FA activity would significantly effect the deadline failure rate performance of the system and would as a result effect any interpretation of differences in the deadline failure rate between the separate models.

The figure 11-1 shows the deadline failure rate of each of the models as the number of tasks at each site is increased from 1 to 12. As the level of competition is increased from 1 to 4 tasks, the EAbA model's performance is significantly worse that of all other models (including the Base model). This results from the EAbA model's use of a pessimistic estimate for EAT_{jk} and ELT_{jk}, resulting in many more task instances being aborted that would have had a chance to complete if allowed to continue for

a longer period of time. As the level of competition is increased from 5 to 8 tasks a significant variation in all the models can be seen. The AbC model is able, under higher competition levels, to improve the deadline failure rate performance over the Base model. All of the AbA models (including the EAbA model) improve their performance over the AbC model. This is a direct result of the extreme benefit from aborting task instances very early under high levels of competition, even to the point that using a pessimistic estimate (in the EAbA model) is the best (for 8 to 12 tasks). However, throughout all levels of competition the LAbA and MAbA models were able to more accurately determine which task instances to abort (within the range of 1 and 4 tasks), and their performance is as good (if only slightly worse) than that of the best performing model.

In the second set of experiments the number of reading task instances was varied between 1 and 21. Each task instance read 256 records from a data file containing 1024 records. The deadlines for the tasks were set to 12ms after the creation time of the task instance. The next arrival of a task instance was then determined to be between 4ms and 25ms after the deadline of the prior task instance. Here again the values chosen for the deadline and next arrival times of the tasks were chosen in order to provide a varying degree of competition in the system that is expected to have the strongest impact on the differences among the models. All task instances in these experiments are also non-critical and essentially sensitive (for the same reasons stated above in the writing task profile experiments).

The figure 11-2 shows the deadline failure rate of each of the models as the number of tasks of each site is increased from 1 to 21. As the number of tasks is increased from 1 to 13 tasks the performance of the EAbA model is significantly worse than all other models, even the Base model. This is a result of the EAbA model's use of a pessimistic estimate for EAT_{jk} (the locking time ELT_{jk} is zero for reading task instances) causing it to abort many task instances that would have had a chance to complete if allowed to continue for a longer period of time. As the level of competition is increased from 15 to 21 the change that was apparent in the Base and EAbA models in the previous experiments for writing profiles, is also apparent for the reading task profiles. For the other models also no significant difference in performance can be seen, compared to the situation seen in the writing task profile (figure 11-1).

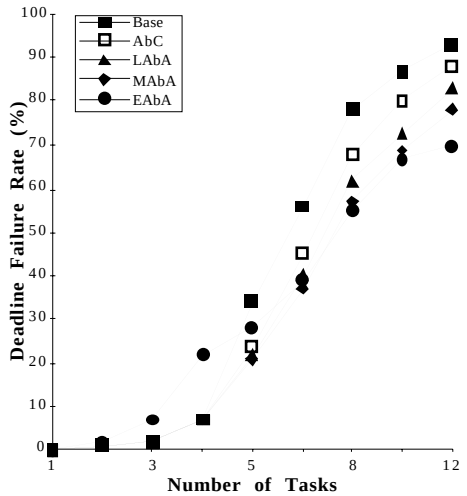


Figure 11-1: Writing Task Profile

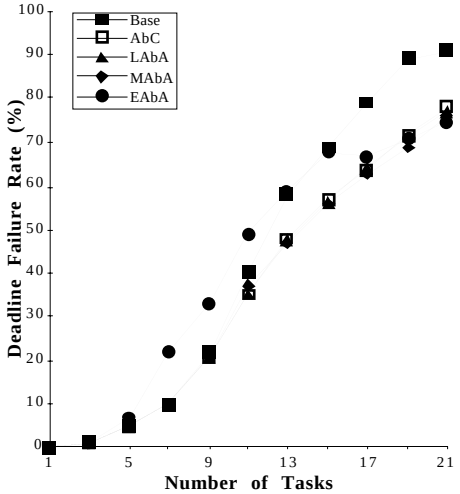


Figure 11-2: Reading Task Profile

In all previous experiments, the writing task profile clearly showed that under high levels of competition there was a significant benefit in aborting task instances as soon as possible. The more complex EAbA, MAbA and LAbA models outperformed the simpler AbC and Base models at least for a higher number of tasks (higher amount of competition for accessing the only file). This benefit resulted directly from the reduced competition for the shared copies of the only file, as task instances with no chance to complete before their deadline were already aborted in the acquisition phase, and therefore as early as possible. Under low levels of competition the performance of the EAbA model suffered significantly from the use of a maximum acquisition duration value for estimating the remaining time required by a task instance. This maximum value (which occurred rather infrequently)

caused the EAbA model at times to abort far too many task instances. Several of these instances if allowed to continue (as in the other models) would have completed before their deadline.

This characteristic tendency of the EAbA model to perform poorly under low levels of competition was also apparent while studying the reading task profiles. Here the cause is solely the variation in the communication times to access the partially remote file copies (no locking overhead). Under higher levels of competition the AbA models again showed significant performance enhancements resulting from the early abortion of task instances (all models were able to outperform the Base model). No significant performance difference could be seen between any of the other models, because of the rather small variation in communication time required to acquire a shadow copy.

In conclusion, both sets of deadline failure rate experiments clearly showed benefits from aborting task instances as early as possible under high levels of competition. Due to the EAbA model using a maximum acquisition duration value (for deciding about aborting a task instance) the deadline failure rate performance degraded significantly under low levels of competition.

11.2 Survivability Performance

The results about the deadline failure rate performance displayed a significant benefit for aborting task instances as soon as possible, under high levels of competition, while delaying the abortion (as in the less pessimistic MAbA and LAbA models) was more beneficial under low levels of competition. However, for safety-critical systems the ability of a system to survive in environments with increasing levels of essentially critical task instances is much more important than the deadline failure rate performance. Therefore, three sets of experiments were setup to test the survivability performance of the models under varying levels of competition and levels of criticality. The sensitivity of all tasks in the following experiments was set within the range of [3..6]. As already mentioned at the beginning of this section the frames of criticality and sensitivity were set to [2...8]. In the following task profiles, task deadlines had then been modeled accordingly to fit tightly. The rather high deadline failure rates (which can be seen in the following figures) were thus to be expected. They are not unacceptable as long as survivability is guaranteed, i.e. the failed task instances had not yet been essentially critical. In this way characteristic differences in the performance of the models under a relatively high load could be more accurately determined.

11.2.1 Low Competition

In a low competition profile, 3 data files with 2 public copies were distributed amongst the nodes. 7 write tasks updated a varying set of two of the three data files. The deadlines for the tasks were set to 28ms, and their next arrival times were between 15ms and 50ms for each task. The criticality of the task profile was varied between the ranges of [7..11], [6..10], [5..9], and [4..8].

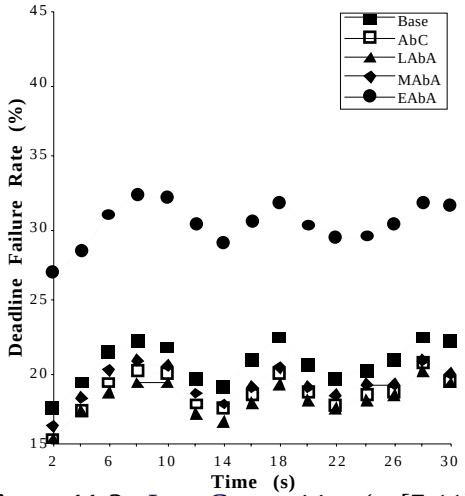


Figure 11-3: Low Competition (C_j [7-11])

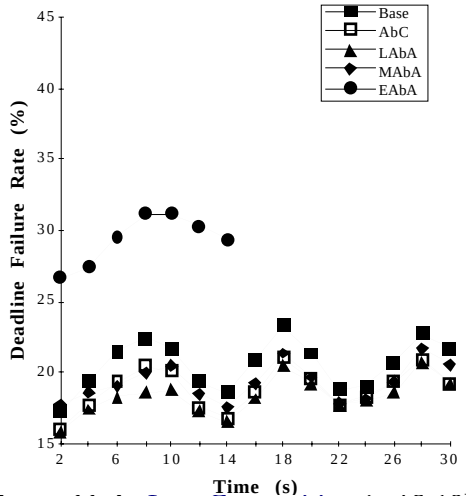


Figure 11-4: Low Competition (C_j [6-10])

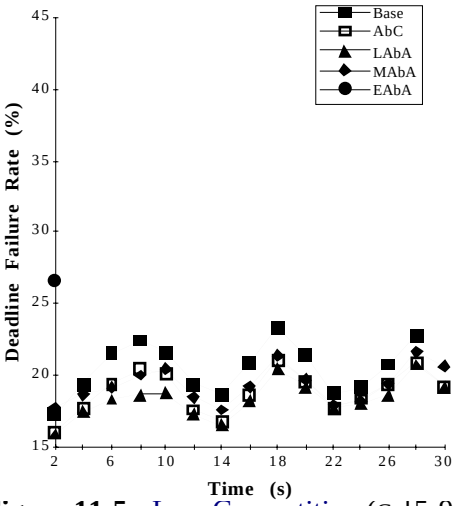


Figure 11-5: Low Competition (C_j [5-9])

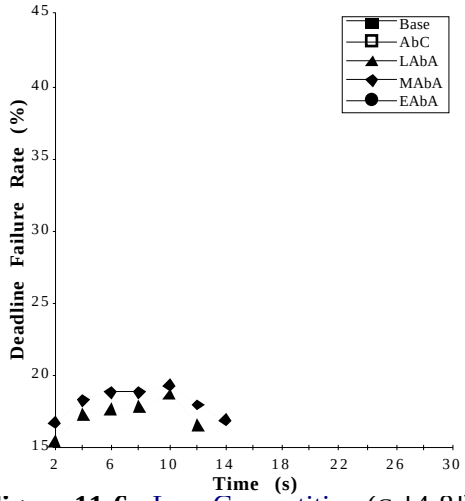


Figure 11-6: Low Competition (C_j [4-8])

For non-critical tasks (criticality range [7..11], see figure 11-3) the performance of the models matched the deadline failure rate performance seen in the previous experiments, as expected. Here all models survive the entire experimental run. It can be clearly seen that the performance of the MAbA model is significantly worse than all of the other models as was shown to be the tendency in the previous experiments. As the criticality was increased to the range of [6..10] (see figure 11-4) the EAbA could no longer complete the entire experimental run (failing after 14 seconds) while all other models successfully completed the entire experiment. The results are the same for the adjacent criticality range [5..9] (figure 11-5). It was only under a very high level of criticality [4..8] (figure 11-6) that all the models failed during the experiment, while both the LAbA and MAbA models were able to survive for 12 to 14 seconds.

11.2.2 Medium Competition

In a medium competition profile 30 tasks updated two of the three data files. The deadlines for the tasks were set between 32ms and 35ms, and the next arrival times for their task instances were set to vary between 60ms and 150ms. The criticality of the tasks was varied between the ranges of [8..12], [7..11], [6..10] and [5..9]. The performance of all of the models for the criticality range [8..12] (non-critical tasks) matched the performance seen in deadline failure rate performance experiments for a medium range of competition (see figure 11-7), with the Base model performing significantly worse than all other models.

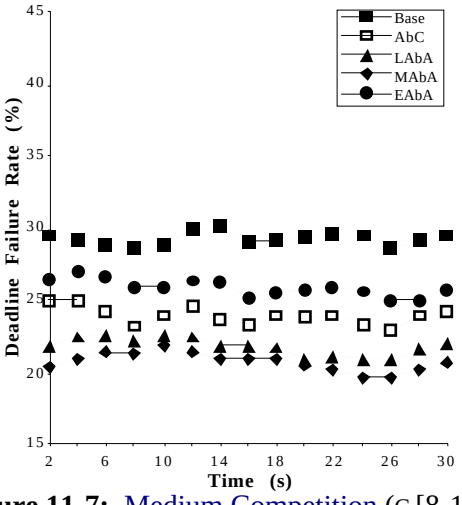


Figure 11-7: Medium Competition (C_j[8-12])

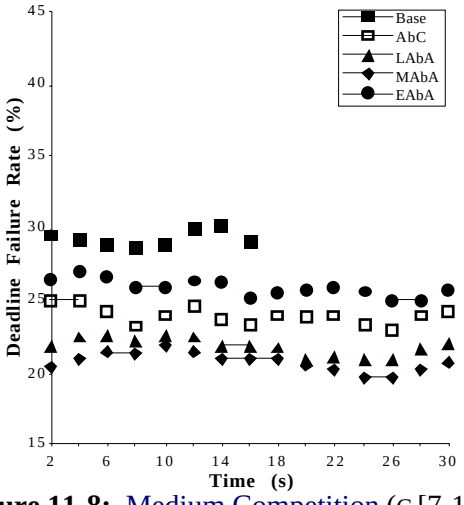


Figure 11-8: Medium Competition (C_j[7-11])

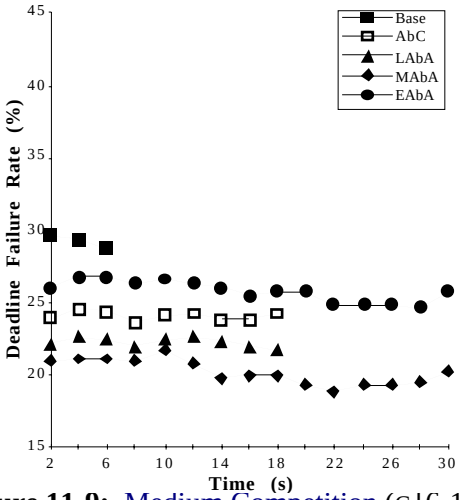


Figure 11-9: Medium Competition (C_j[6-10])

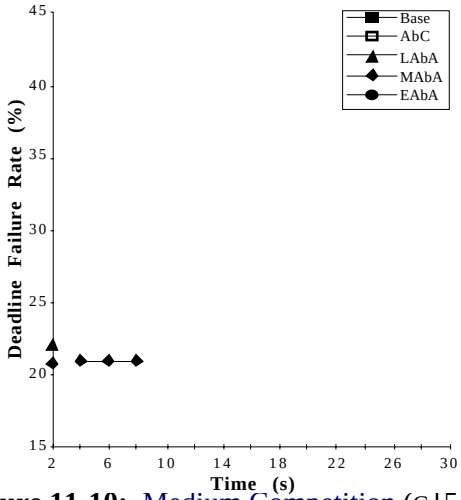


Figure 11-10: Medium Competition (C_j[5-9])

However, as the criticality was increased into the frame [7..11] (see figure 11-8) the Base model could no longer complete the entire experiment and failed after 6 seconds. The AbC and LAbA models failed after 18 seconds. Both the MAbA and EAbA models were able to complete the entire experiment. Under high levels of criticality, here given by the frame [6..10] (see figure 11-9) all models failed during the experiment while both the LAbA and MAbA models were able to survive for 2 and 8 seconds, due in part to their superior deadline failure rate performance and the resulting fewer essentially critical task instances. It was only under a very high level of criticality [5..9] (figure 11-10) that all the models failed during the experiment, while the MAbA model was able to survive for 8 seconds.

11.2.3 High Competition

In a high competition profile 45 tasks updated two of the three data files. The deadlines for that tasks were set between 32ms and 35ms, and their next arrival times for subsequent task instances were set to vary between 60ms and 150ms. The criticality of the tasks was subsequently varied within the ranges of [8..12], [7..11], [6..10] and [5..9]. The deadline failure rate performance of all of the models for the criticality range [8..12] matched the performance seen in deadline failure rate performance experiments for high competition (figure 11-11), with the Base model performing significantly worse than all other models. As the criticality was increased into the frame [7..11] (see figure 11-12) the Base and AbC models could no longer complete the entire experiment and failed after 2 and 6 seconds, respectively. All of the other models successfully completed the entire experiment. As the criticality was increased into the frame [6..10] (see figure 11-13) the Base and AbC models could no longer complete the entire experiment and failed after 2 seconds. The LAbA model failed after 12 seconds, while the MAbA model was able to survive for 18 seconds. The EAbA model was able to complete the entire experiment. As the criticality was again increased into [5..9] (figure 11-14) only the MAbA and EAbA models were able to survive for some period of time (8 seconds and 16 seconds, respectively). This significant performance enhancement with respect to survivability is again the result of a decreased level of competition resulting from increased number of task instances aborted by the two models.

In terms of survivability the results of the experiments clearly show a performance benefit under higher levels of competition which goes along with the early abortion of task instances (see particularly figure 11-13 and figure 11-14). Although the deadline failure rate performance of the EAbA model was fairly poor compared to both the AbC and LAbA models, the first model was able to outperform the latter ones regarding survivability, under both the medium and high competition profiles. The reason is that the increased number of aborted task instances in EAbA causes a significant reduction regarding the degree of competition. The decreased competition reduces the acquisition times thus making it easier for essentially critical task instances to obtain there required locks. This was particularly

apparent in the medium competition profile of figure 11-9. Here the EAbA model's deadline failure rate was very poor but its survivability performance was much better than for the other models with better deadline failure rate performance. However, the early abortion of tasks in EAbA becomes detrimental when the level of criticality is very high, due to the increased number of aborted task instances (and subsequently increased number of essentially critical task instances). This is clearly displayed in figure 11-10 where the EAbA model fails very early. Instead, the superior deadline failure rate performance of the LAbA model allows it to survive slightly longer. Finally, the early abortion of task instances was clearly very beneficial to both the MAbA and EAbA under very high levels of competition (figure 11-13 and figure 11-14). Here the AbC and LAbA models exhibit their inability to handle this degree of competition while both the EAbA and MAbA models' survivability performance was clearly superior to the performance of the other models.

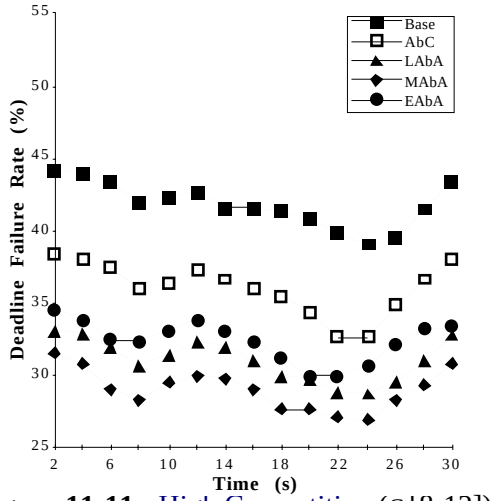


Figure 11-11: High Competition (C_j[8-12])

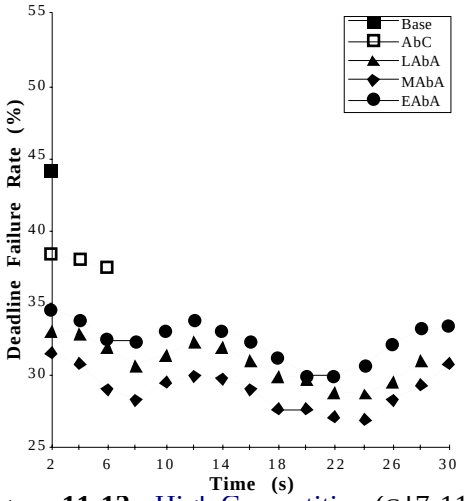


Figure 11-12: High Competition (C_j[7-11])

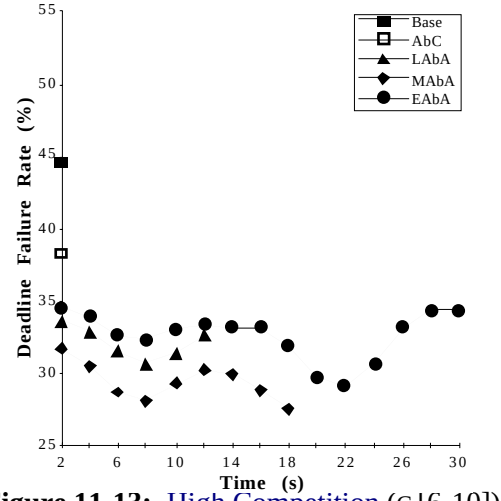


Figure 11-13: High Competition (C_j[6-10])

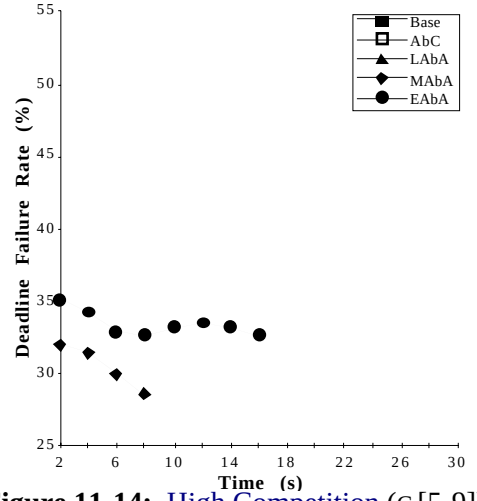


Figure 11-14: High Competition (C_j[5-9])

Under low levels of competition the effect of using a maximum value to determine the remaining acquisition time caused the EAbA model to perform very poorly, in terms of deadline failure rate (see figure 11-3, figure 11-4 and figure 11-5) and survivability (see figure 11-4 and figure 11-5). The reason is that in EAbA the measure to abort task instances during their acquisition phase is the maximum acquisition duration, taken over the previous 10 instances. This is an extremely pessimistic policy of early abortion which in the worst case may result in a constant high duration assumption for 10 subsequent task instances. A high number of task abortions, however, has no significant influence on the competition given that there is a small number of competing task instances anyway in the system. A similar argument explains why the survivability performance of the excessively optimistic LAbA model (using the minimum acquisition duration) was very poor under high levels of competition (see figure 11-14).

MAbA performs very well, both in terms of survivability and deadline failure rate, on all levels of task competition as well as for all ranges of task criticality. It outperforms all other models regarding the deadline failure rate performance for medium and high competition (see figure 11-7 through figure 11-14), and it represents the best survivability strategy for low and medium competition (see figure 11-3 through figure 11-10). For a low competition profile the deadline failure rate is very close to the optimal LAbA strategy (see figure 11-3 through figure 11-6), and for a high competition MAbA survives nearly optimally (see figure 11-11 through figure 11-14). The reason is that the measure for aborting task instances during the acquisition phase is the average acquisition duration value that most flexibly adapts to changing locking and communication times stemming from a varying competition of task instances. Since the competition is excessively "reduced" through the EAbA model by aborting task instances not yet essentially critical, the latter model has a slightly better survivability performance than MAbA under a high competition. Similarly the late abortion under LAbA gives this model a small advantage over MAbA under a low competition, regarding the deadline failure rate. Given the wide range of task competition and criticality the MAbA model is clearly the best choice and therefore was choosen as the task monitoring policy implemented in MELODY. As stated before, for a better judgement of survivability up to 30 runs were performed but no significant differences were found in comparision to the smaller number presented.

Chapter 12 Conclusion and Future Outlook

The MELODY project evolving from an extensive Ph.D. research study in 1988 [Ali88] has currently progressed through six phases (see section 1.2). Phase 6 constituted a major model extension, from *simulation* to ***distributed experiments*** that explicitly reflected actual task computation, file manipulation and real communication traffic. Upon completion of phase 5, initial studies regarding the integration of task and resource scheduling had been conducted, however, the insights had been limited to investigating task profiles where survivability was not at stake. At this point no work had been done to integrate either of the File Assigner or Run-Time Monitor activities. Server integration however needed to be predictable in order to deal with the real-time environment, while, at the same time needing to be adaptable in order handle the unpredictable environment typical of safety-critical real-time systems. In the following paragraphs the specific achievements accomplished during phase 6 are detailed.

In previous development phases, experiments had been conducted to compare the functionality of MELODY's file system to simpler yet less flexible models. These simpler models (Public and Private) exhibited some but not all of MELODY's functionality. Previous simulators however had not been able to realistically implement task computation, file manipulation and computation. As the project moved to a distributed implementation it was thought necessary to replicate these experiments (see chapter 8). The reason being that previous results could be utilized to validate the distributed implementation, while at the same time the effects that real task computation, file manipulation and communication would have on the performance of the MELODY model could also be obtained. This had been an open question upon completion of phase 5. The file system functionality experiments clearly showed no significant change in the characteristic tendencies between the three models. There were however significant deadline failure rate performance changes, but at no time did this change any of the model tendencies. As in simulation, it was shown that MELODY's adaptive features pay off despite the increased overhead resulting from the more complex model.

In phase 5, policies for integrating task and resource scheduling had been developed. These policies were developed to handle the conflicting goals of tasks needing to be scheduled as early as possible with the need to reduce the overhead caused by Task Scheduler activities, and at the same time providing a wiser scheduling decision based on increased numbers of waiting task instances. Upon completion of phase 5, initial simulation experiments had been done on these integration policies. These experiments however had only evaluated the performance of the policies in environments were survivability was not at stake. To obtain a full understanding of the performance, extended experiments were conducted to evaluate them under wide ranges of criticality, sensitivity and read/write dominance (see chapter 9). Again, the results did show some significant deadline failure rate performance changes, due to the distributed environment, but there were no changes in the characteristic model tendencies. The Dynamic model clearly outperformed all other models both in non-critical environments (as was shown in simulation studies) and environments were survivability was at stake. It was therefore chosen as the Task Scheduler/File Server integration policy implemented in MELODY.

A trade-off has to be made, under changing requests and deadline failure patterns, between the costs of serving file requests with a given distribution of files and the cost for realizing various alternative distributions. In MELODY, the File Assigner is responsible for adaptively changing the distribution of files according to the changing and typically unpredictable environment in order to improve the system's performance. At the end of phase 5, no work had been done to evaluate the effect that File Assigner invocation would have on the system's performance. Distributed experiments were conducted to evaluate the new novel integration policies under wide ranges of criticality, sensitivity and

read/write dominance (see chapter 10). The integration policies were also compared against a base-line model in which the File Assigner was deactivated. In all experiments, the deadline failure rate performance of the File Server-Oriented integration model clearly outperformed both the Task Scheduler-Oriented integration and base-line models. In terms of survivability, the File Server-Oriented integration model was able to match the performance of the Task Scheduler-Oriented integration model, and both models clearly outperformed the base-line model. The File Server-Oriented integration model was therefore chosen as the model to be implemented in MELODY.

The policy to reverse the order of task and resource scheduling in MELODY had been established in phase 5. This had been done to abort tasks as early as possible and at the same time lock resources as late as possible. As a result, blocking of remote task instances caused by a task instance that held a resource lock but may not be scheduled would be eliminated. The reversed order however also results in the Task Scheduler no longer being able to guarantee that a task instance would meet its deadline. It was hoped that this lack of a guarantee would be more than made up for by the lower amount of remote blocking by task instances that would not complete prior to their deadlines. A novel Run-Time Monitor module was introduced in phase 5 to supervise resource acquisition and abort task instances as early as possible. However, no work had been done to develop policies for how task monitoring would be integrated into MELODY. The effect on the system's performance resulting from the overhead caused by task monitoring was an open question. A major accomplishment of this thesis is the development and evaluation of policies for task monitoring in MELODY (see chapter 7). Four increasingly pessimistic policies were developed. Experiments conducted on the various policies were done to evaluate their performance under wide ranges of criticality and task competition levels (see chapter 11). The results showed that the extreme models (*Earliest Abort before Acquisition* (EAbA) and *Latest Abort before Acquisition* (LAbA)) were able to slightly outperform the *Medium Abort before Acquisition* (MAbA) model under extreme competition levels, while the more adaptive MAbA model, through its use of average acquisition times, performed very well across all ranges of criticality and levels of task competition. Due to this performance result, the MAbA model was selected as the task monitoring integration policy implemented in MELODY.

An objective of this thesis was to develop and apply the novel **Incremental Experimentation** methodology to implement MELODY as an academic prototype of a distributed real-time safety-critical system. Given that no closed-form solution can be obtained under the orthogonal objectives of real-time responsiveness and reliability, this was instrumental for taking into account the large number of relevant implementation and integration parameters, and their complex interdependencies, while at the same time designing the MELODY system in a transparent procedure. Incremental experimentation allowed us to start with a coarse system model (with accurate logical expectations regarding its behavior). Through experimental investigation, these expectations were validated. If they were found to successfully stand the tests, extended expectations or model features could be generated for refining the previous design model (as well as its performance criteria). The refinements are done in such a way that the previous experimental configurations are extreme model cases or data profiles that both logically and experimentally are expected to reproduce the behavior of the previous model. If the special experiments had the expected results, the performance aspects (or tendencies) under fully varying parameters could unambiguously be attributed to the influences of the refined model features. In this way, all relevant design and analysis aspects could be integrated in a systematic way into a realistic and pragmatic modeling methodology, while at the same time reconciling the conflicting issues.

While in the previous phases simulation was used, a distributed implementation of MELODY's modules and services was done in phase 6. In this light, the File Assigner and File Server functions needed to be reevaluated. It was not a real application environment that served as a testbed, but a wide variety of task and data profiles were created for covering a reasonable range of application circumstances. The evaluation method was to compare the proposed mechanisms and services with simpler ones that could be expected to have considerably smaller overhead. During the comparative experiments it was found throughout that the ensuing advantage of the simpler mechanisms was easily

outweighed by the higher flexibility and adaptivity of the more sophisticated MELODY functions.

The results from phase 6 can be used to refine and enhance the MELODY model for including more complex issues regarding safety-critical real-time systems. These extensions will require more complex parameters and functions be incorporated into MELODY. The evaluation of such model extensions will only be carried out using the incremental experimentation methodology based on the results presented in this thesis.

Future plans for the MELODY project include expanding the integration methods presented. This includes developing techniques that would allow the parameters to automatically adapt to changes in the environment. For example, the parameters for Task Scheduler/File Server integration could be made adaptable to the various task profiles encountered. Remember that tasks in safety-critical real-time systems not only have to meet their associated deadlines, but most of these are critical in the sense that the system would not survive in the case of a certain number of deadline failures of subsequent task instances. A task instance in such a critical stage would have a hard deadline, and is said to have become **essentially critical** (see section 1.1). When high levels of these nearly essentially critical tasks (if this task instance fails the next instance will become essentially critical (see section 3.1.1)) are encountered the Task Scheduler could be invoked earlier to schedule these task instances as soon as possible. Policies for task monitoring could also be refined to vary the estimate used to abort task instances based on the criticality of the task instances. In this way nearly essentially critical task instances would be aborted under the most optimistic LAbA model, while non-critical task instances could be aborted using the most pessimistic EAbA model. The effect of these refinements on the system's performance however could only be safely attributed to the enhancements by comparing them to the well-understood results presented in this thesis.

It is conceived that in phase 7 of the MELODY development refinements to criticality and sensitivity would be undertaken to make the system more reactive to changes in these aspects. In this way, the system would not only perform emergency actions following the failure of a nearly essentially task instance, but would take action earlier in order to prevent a task instance from becoming essentially critical.

The MELODY model could also be extended to include the concept of similarity [KuM91 and MCG98] in order to relax the consistency requirements of real-time transactions. Typically in safety-critical real-time systems, there are sensors at the periphery of the application system that input a continuous stream of data into preprocessors where tasks filter and digitize this data. For each data object (which is assumed to be a template for its instances) a similarity bound is assigned, which may be reset according to environmental circumstances (typically the bound would be lowered as the environment gets more unpredictable). Only if for a data object (the instances of which are filtered out at a particular preprocessor) the next instance is no longer similar, messages would be sent to alert the invocation of control tasks at sites which receive such object instances, in order to possibly take corrective actions. Each control task is also assigned a similarity bound (also adjustable to the unpredictable environment) for each object/resource that it needs to access. When a message about exceeding the similarity bound is received at a site that holds the control task, the similarity bound would be compared against the measure of the change to the data object (included in the message). If the change exceeds the similarity bound, the control task will be invoked. If the similarity bound is not less than the change values for all objects accessed by the control task, the control task will not be invoked (this is an explicit model for setting up the principle of typically aperiodic tasks in safety-critical systems).

Control tasks in safety-critical environments would be characterized by: a deadline, criticality, sensitivity, and a similarity predicate. Note that sensitivity and similarity are closely related concepts in the MELODY context (they both refer to the difference between latest and earlier data/file information). However, they are still independent parameters. As the situation for a system becomes less predictable, sensitivity will be more relaxed (see section 3.1.2), while similarity bounds instead are

more tightly approached. Another difference is that sensitivity is a measure that directly relates to task management while similarity reflects the data management, and is directly environment-oriented. Given that similarity predicates are defined (and adjusted) for tasks, transactions, as well as for files, similarity could be used at different levels in MELODY. If a change to the public copies of a file is within the similarity bounds, the operation to refresh the private copies of the file need not be executed at this time. Also, as described previously, when changes to a file are similar to the previous ones in the preprocessor, tasks that operate on these files (including those which write/transfer these values/ structures to processing sites) would not be invoked. Transactions for safety-critical applications would be characterized by setting the four parameters (given above) for the subtransactions on the leaf level (which are tasks), or they could be derived from values assigned to the transaction itself.

The performance, reliability and availability of an application can be enhanced through the replication of data on multiple sites. But recovery and concurrency control problems are aggravated by distribution and replication. Three primary techniques for distributed concurrency control have been lately proposed in the literature: locking, validation (optimistic) and timestamping [XRH98 and XSS98]. However, only the first two techniques are suited to unpredictable real-time environments since timestamp-based protocols, due to fixing the transaction commit order *a priori*, appear fundamentally ill-suited. In the locking protocol, a transaction that intends to read a file has only to set a read lock on any copy of the item; to update an item, however, write locks are required on all copies. Write locks are obtained as the transaction executes, with the transaction blocking write requests until all of the copies have been successfully locked. Locks are held until the transaction either has committed or aborted. In the optimistic protocol, the execution of a transaction consists of three phases: read, validation and write, where validation is a distributed two-phase process to coordinate validation of all updates. The key component is the validation phase where the outcome of a transaction is decided. If any update at a site fails during the validation phase, the transaction will be aborted, otherwise the transaction will go ahead and commit the update in the third phase. Concurrency control algorithms enhanced by features relating to the safety-critical parameters (given in the previous paragraph) would be much more efficient for safety-critical applications than those which just take into account deadlines.

In phase 7, the MELODY model will be refined and enhanced to handle the more complex issues regarding safety-critical real-time systems. Particularly, emphasis will be placed on model enhancements to handle similarity and concurrency control in safety-critical systems and reactive databases. These extensions require that more complex parameters and/or more complex functions be incorporated into MELODY. The evaluation of which could only be safely carried out using the incremental experimentation methodology. Changes to the performance of MELODY could be attributed to the new model extensions, while the results present here would be modeled as extreme parameter settings. This refers to ongoing mid-range project work, in cooperation with the University of Massachusetts (Amhurst), University of Texas (Austin), and the Indian Institute of Technology Bombay.

Appendix

Appendix A Communication Model

MELODY provides both UDP and TCP communication protocols for the handling of communication between MELODY nodes. The UDP protocol provides interprocess communication to service messages that are characterized as either control messages (Delayed Insertion Protocol) or informative (relocation/ replication/ deletion of file copies). The TCP protocol provides for a reliable communication for both file transfer and data transfer. More information regarding specifics of either UDP or TCP protocols can be found in [Rag93, Ste94a and Ste94b]. The following is only a description of the model and procedures used by MELODY to provide communication services. The underlying network protocol is not specified by MELODY and is envisioned to be determined by the hardware implementation.

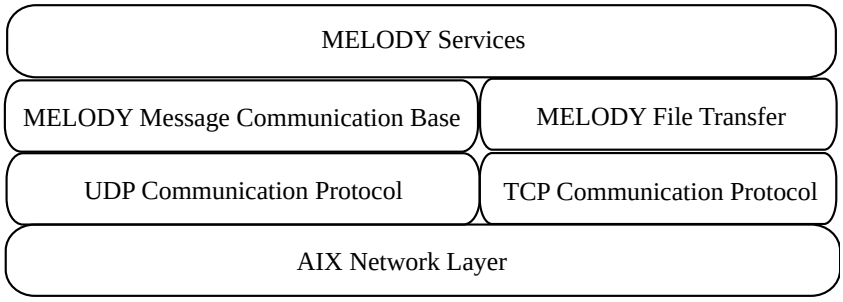


Figure A-1: MELODY Communication Layers

A.1 Message Communication

The MELODY communication model allows for messages to be sent using three different methods: point-to-point, broadcast and multi-cast. Point-to-point communication provides communication between two specific nodes or more generally between two specific processes. Broadcast communication is implemented in order to allow MELODY services to send requests to a certain service (process) at all other MELODY nodes. A message that is sent using broadcast is received by every node in the system including the sending node. Multi-cast communication allows for a service to send a message to a select group of services within the system.

Due to the use of UDP as protocol for message sending, two aspects of communication in MELODY had to be modified. UDP does not provide for a multi-cast within the protocol. Therefore, the implementation utilizes a broadcast and requires the receiving service to determine if the message is relevant or not. UDP as a communication protocol does not guarantee that any message sent arrives at its destination. As a result, services must handle the possible loss of messages due to the unreliable communication medium. The main advantage of UDP however is the speed at which messages can be sent from site to site. This time logically increases as the number of other sites trying to communicate at the same time increases. Broadcast communication, by definition, is slower due to the need of a message to be handled at more that one site.

A.2 File Transfer

File Transfer within the MELODY communication model is designed in order to handle both files and large amounts of data transfer between to sites. The transferring of files utilizes a client/server approach to handling the communication. The client executes the algorithm A-1 to receive a file, while

the server executes algorithm A-2 to send the file. Utilizing TCP communication in MELODY has the distinct disadvantage that a connection must be established prior to the sending of the file. However, once the connection is established actual communication time, to send the contents of the file, is much faster than with UDP.

```
receive-file(file F)
{
   accept connection from sending-host for file F
   create file F
   while receive portion P of file F {
         write portion P to file F; }
   close file F
}
```

Algorithm A-1: Receive File

```
send-file(file F)
{
   connect to receiving-host for file F
   open file F
   while read portion P of file F {
         send portion P to receiving-host; }
   close file F
}
```

Algorithm A-2: Send File

Appendix B Time Synchronization

The notion of time plays an important role in any real-time system. In MELODY, time is even more important due to task instances requiring that certain operation be completed prior to a hard real-time deadline. Without an accurate time synchronization protocol inconsistencies may occur in that a task requesting an operation on a remote node may believe that the task would complete before it's associated deadline, while the remote node would have aborted the operation due to a local clock that was not synchronized (and therefore ahead of the local node). Due to the special requirements of safety-critical real-time operating system (especially in concern with essentially critical task instances) MELODY can not allow the local clocks at two or more sites to be considered inconsistent (the variance in the clock times not within a predetermine interval). There has been much research into developing protocols that would guarantee that clocks within a distributed environment were synchronized including the following well known protocols:

4.3 BSD Time Protocol:This protocol is the UNIX standard protocol to synchronize clocks within a LAN environment [Ste92]. This protocol guarantees that the variance between any two node clocks is no more than 20 milliseconds.

GPS Based Time Server:The global positioning system (GPS) is a world-wide satellite navigation system [WaH95]. GPS is capable of determining the position of a node within a tolerance of 100 meters, and receiving the an official world Universal Time Coordinate (UTC). This protocol guarantees that the variance of any GPS receiver clock is no more than 100 nanoseconds. However, to set the local nodes clock may cause the variance to be increased.

However, due to the impreciseness of the BSD protocol and the special hardware requirements of the GPS protocol MELODY has developed it's own time synchronization protocol that has been defined and validated in extensive experiments [Seg97]. The time synchronization protocol has the following characteristics:

1: Maximum clock variance between any two node is less than 100 microseconds.
2: The preciousness of the local clock is independent of the overhead of the local site.
3: The overhead caused by synchronization of the local clock is minimal and has no significant effect on the survivability of task instances within the system.

The time synchronization protocol is based on the idea that the variance between any two local clocks changes at a constant rate [Tan95]. In extensive experiments [Seg97] this proved to be the case for the IBM RS6000 machines on which the implementation has been completed. For the seven machines tested the variation in the change of the clocks was no greater than 50ms (figure B-2). Since the variance between any two nodes clocks varies at a fairly constant rate, local nodes resynchronize their clocks based on the known constant change in the variance between the local node and a chosen time server. Therefore, clocks are only synchronized upon initialization of the local node. To synchronize the clocks within the distributed environment a node is required to utilize the following protocol in order to ensure that it's local clock remains within an tolerable clock variance. The time synchronization protocol (figure B-3) can be separated into two distinct portions. The initial synchronization with a predetermined time server, and the resynchroniztion based on a constant variance.

1: Determine location of the node that has been selected to serve as the time server within the MELODY distributed environment. Once this server is found proceed to step 2.
2: Request the time stamp from the time server by sending a time request message to the time server. Upon receiving the time stamp message proceed to step 3.
3: Determine is the message received from the time server has been received within a message delay window. If the message has not been received within this window then stop the synchronization and return to step 2. If the message has been received within the window then proceed to step 4.
4: Determine the difference in the local clock and the time server clock by using following equation $U(T_{local}) = T_{server} + (D_{message}/2) - T_{local}$. Where T_{server} is the time stamp returned by the time server, T_{local} is the time stamp on the local node, and $D_{message}$ is the communication delay required to determine the server's time stamp. If the variance of the local clock is greater that the maximum

allowed value then return to step 2. Otherwise, the clocks are within a tolerable clock variance, proceed to step 5.

5: Wait for a given interval of time T_{sleep} elapse before proceeding to step 6.

6: Resynchronize the local clock using a known local constant clock variance. Return to step 5.

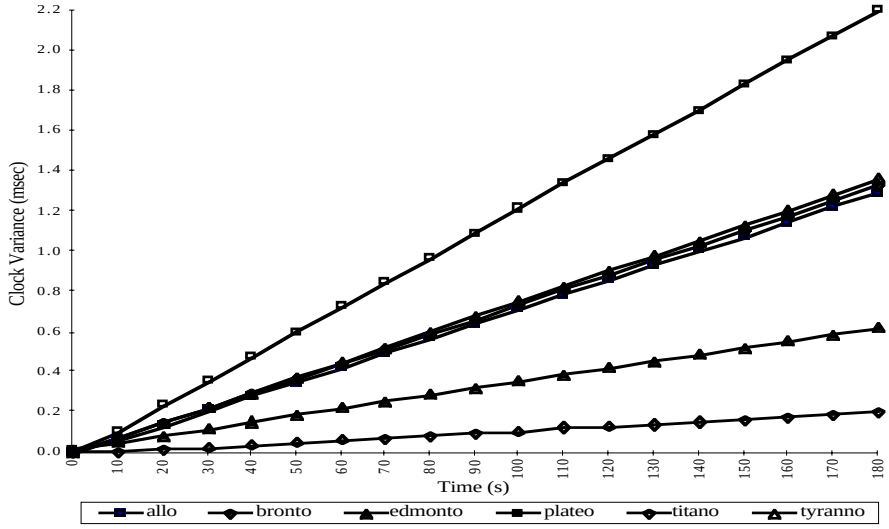


Figure B-2: Clock Variance without any Time Synchronization

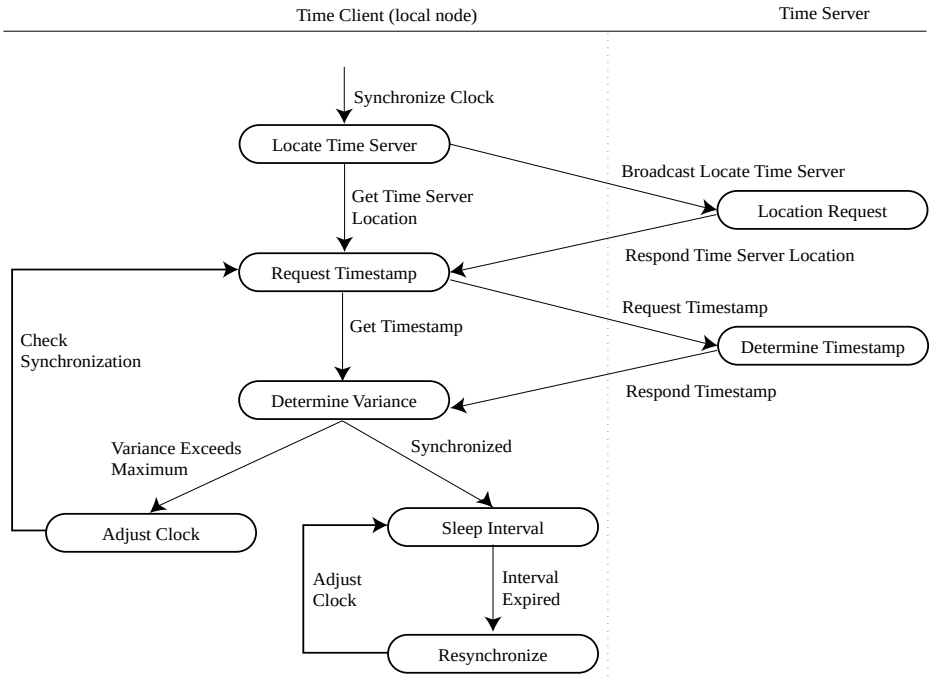


Figure B-3: MELODY Time Synchronization Protocol

The experimental results shown in that the over a significant interval (180 seconds) there is no significant change in the clock synchronization. For each of the six nodes it can be seen that after an initial synchronization the clocks do not vary more than 10 microseconds. The time points shown in the charts points taken immediately following the resynchronization of the clocks. The actual clock variance may have been greater than this but would never had exceeded the following variance plus the normal clock drift shown in figure B-2.

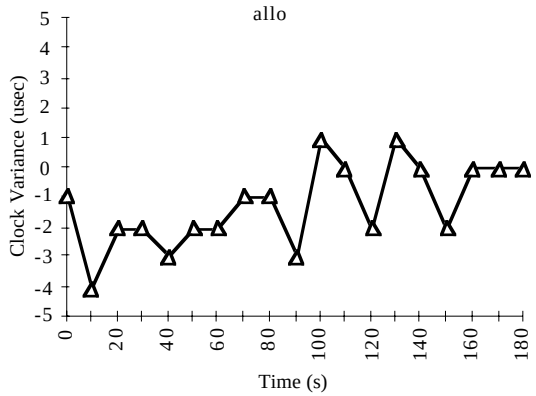


Figure B-4: Synchronization (Allo)

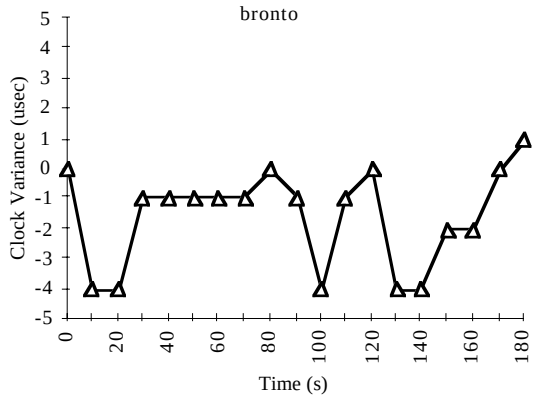


Figure B-5: Synchronization (Bronto)

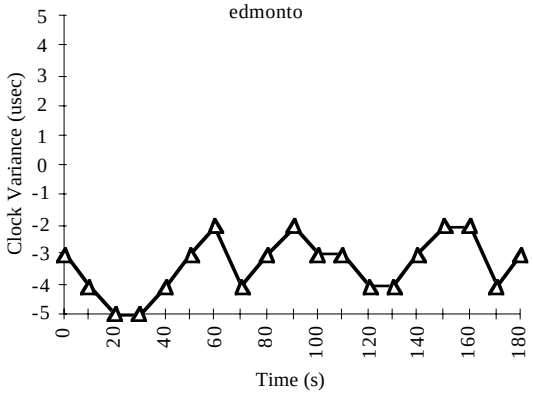


Figure B-6: Synchronization (Edmonto)

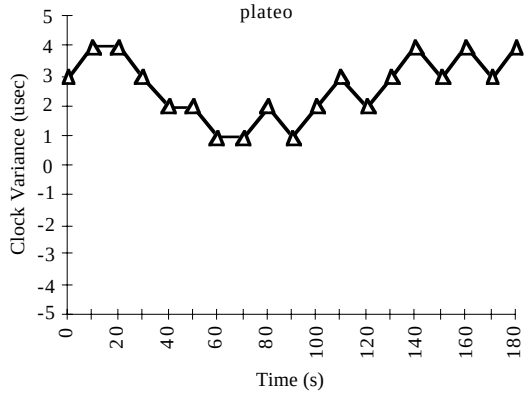


Figure B-7: Synchronization (Plateo)

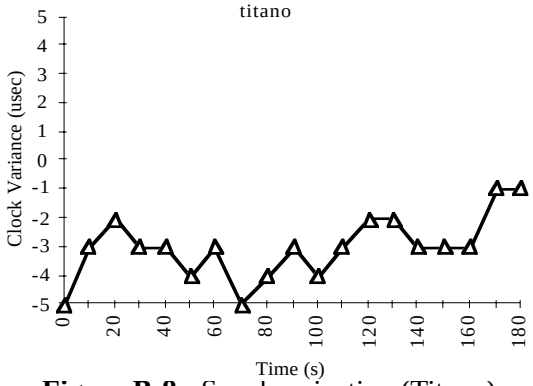


Figure B-8: Synchronization (Titano)

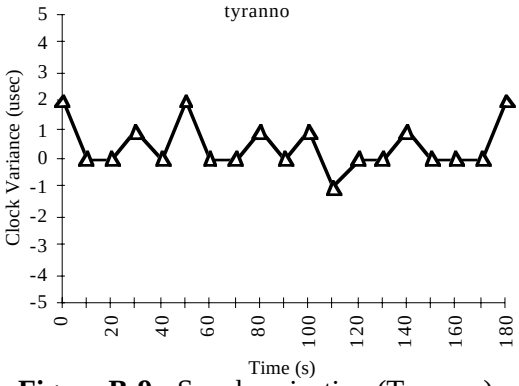


Figure B-9: Synchronization (Tyranno)

Appendix C Source Code

MELODY is implemented is a homogeneous distributed environment. The experiments were conducted on a Token-Ring network with 7 IBM RS6000 machines. The implementation has been highly integrated with the AIX operating system (AIX version 4.2.1) using the programming language C. Here MELODY is implemented by using the AIX kernel functions. It's commands are assigned the highest possible priority in AIX. Communication between machines was handled using TCP and UDP protocols.

The source code for the complete MELODY implementation can be obtained from Lehrstuhl III (operating systems and computer architecture) at the University of Dortmund Germany. All of the source code, scripts and experiment scripts can be found on the machine ls3.cs.uni-dortmund.de. All source code and the Makefile used to build the executables are contained in the directory /melody/melody/source. The scripts required to start the MELODY servers and perform individual experiments can be found under the directory /melody/melody/bin. All scripts used to conduct the experiments found in this dissertation can be found under the directory /melody/melody/experiments.

References

[ABD95] Audsley N.C., Burns A., Davis R.I., Tindell K.W., Wellings A.J. (1995); "Fixed Priority Scheduling: An Historical Perspective"; *Real-Time Systems;* Vol. 8, 1995

[Ali88] Alijani G.S. (1988); "Object Mobility in Distributed Computer Systems"; *Ph.D. Dissertation*; Wayne State University

[Alt95] Altenbernd P. (1995); "Allocation of Periodic Hard Real-Time Tasks"; *Proceedings of the IFAC/IFIP Workshop on Real-Time Programming*; (1995)

[ATB93] Audsley N.C., Tindell K., Burns A. (1993); "The End of The Line for Static Cyclic Scheduling?"; *Proceedings of the 5th EUROMICRO Workshop on Real-Time Systems;* (1993)

[CJD91] Chodrow S.E., Jahanian F., Donner M. (1991); "Run-Time Monitoring of Real-Time Systems"; Proceedings of the 12th *IEEE Real-Time Systems Symposiums;* December 1991

[ChL87] Chu W.W., Leung K.K. (1987); "Module Replication and Assignment for Real-Time Distributed Processing Systems"; *Proceeding of the IEEE*; Vol. 75

[Dan92] Daniels D.C. (1992); "The Design and Analysis of Protocols for Distributed Resource Scheduling under Real-Time Constraints"; *Ph.D. Dissertation*; Wayne State University

[DoF82] Dowdy L.W., Foster D.V. (1982); "Comparative Models of the File Assignment Problem"; *ACM Computing Surveys* Vol. 14 No. 2 (1982)

[FuG91] Furth B., Grostick D., Gluch D., Rabbat G., Parker J., McRoberts M. (1991); *REAL-TIME UNIX SYSTEMS, Design and Application Guide*; Kluwer Academic Publishers, 1991

[HaL96] Ching-Chih H., Kwei-Jay L., Chao-Ju H.; "Distance-Constrained Scheduling and Its Applications to Real-Time Systems"; *IEEE Transactions on Computers*, Vol. 45, No. 7 (1996)

[HaS89] Haben D., Shin K.G. (1989); "Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times"; *IEEE Real-Time Systems Symposium* (1989)

[HaS90] Haban D., Shin K. (1990); "Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times"; *IEEE Transactions on Software Engineering*; Vol. 16 No. 12 (1990)

[HaS91] Wolfgang A. Halang, Alexander D. Stoyenko (1991); *Constructing Predictable Real Time Systems*; Kluwer Academic Publishers, 1991

[HaW90] Haben D., Wybranietz D. (1990); "A Hybrid Monitor for Behaviour and Performance Analysis of Distributed Systems"; *IEEE Transaction on Software Engineering;* Vol. 16, No. 2 (1990)

[Hav68] Havender J.W. (1968); "Avoiding Deadlocks in Multitasking Systems"; *IBM Systems Journal*; Vol. 7, No. 2

[HCM92] Haritsa J.R., Carey M.J., Livny M. (1992); "Data Access Scheduling in Firm Real-Time Database Systems"; *Real-Time Systems*; Vol. 4 No. 3 (1992)

[HKM97] Ho S.J., Kuo T.W., Mok A.K.(1997); "Simularity-Based Load Adjustment for Real-Time Data Intensive Applications"; *Proceedings of the 18th IEEE Real-Time Systems Symposium*, San Francisco, California, December 1997

[HLF95] Hsueh C.W., Lin K.J., Fan N. (1995); "Distributed Pinwheel Scheduling with End-to-End Timing Constraints"; *IEEE Real-Time Systems Symposiums;* December 1995

[JeN90] Jensen D.E., Northcutt J.D. (1990); "Alpha: A Non-Propriety OS for Large, Complex, Distributed Real-Time Systems"; *Proceedings of the Second International IEEE Workshop on Experimental Distributed Systems*, Huntsville, Alabama, 1990

[JRR94] Jahanian F., Rajkumar R., Raju S.(1994); "Runtime Monitoring of Timing Constraints in Distributed Real-Time Systems"; *Real-Time Systems*; Vol. 7 No. 3 (1994)

[KaR95] Kamath M.U., Ramamritham K.(1995); "Performance Characteristics of Epsilon Serializability with Hierarchical Inconsistency Bounds"; *International Conference on Data Engineering;* December 1995

[KuM91] Kuo T.W., Mok A.K. (1991); "Load Adjustment in Adaptive Real-Time Systems"; *Proceedings of the 12th IEEE Real-Time Systems Symposium*, December 1991

[KuM93] Kuo T.W., Mok A.K.(1993); "SSP: A Semantics-Based Protocol for Real-Time Data Access"; *Proceedings of the 14th IEEE Real-Time Systems Symposium*, December 1993

[LaH97] Lam K., Hung S. (1997); "Preemptive Transaction Scheduling in Hard Real-Time Database Systems"; *Journal of System Architechture*; Vol. 43, No. 9 (1997)

[LeS90] Levy E., Silberschatz A. (1990); "Distributed File Systems: Concepts and Examples"; *ACM Computing Surveys* Vol. 22 No. 4 (1990)

[LeM78] Levin K.D., Morgan H.L. (1978); "A Dynamic Optimization Model for Distributed Database"; *Operation Research;* Vol. 26, No. 5 (1978)

[LeM80] Leung J.Y.T., Merril M.L. (1980); "A Note on Preemptive Scheduling of Periodic Real-Time Tasks"; *Information Processing Letters* Vol. 11, No. 3 (1980)

[LeW82] Leung J.Y.T., Whitehead J. (1982); "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks"; *Performance Evaluation;* Vol. 2, No. 4 (1982)

[LiL73] Liu C.L., Layland J.W. (1973); "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment"; *Journal of the Association for Computing Machinery*; Vol. 20

[Lin93] Lind J.A. (1993); "Dynamic Integration Policies for Distributed Task and Resource Scheduling in Mission-Critical Systems"; *Masters Thesis*; Wayne State University

[LiS90] Lin K.J., Son S.H. (1990); "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order"; *Proceedings of the 11th IEEE Real-Time Systems Symposium*; December 1990

[LLL92] Liu J.W.-S., Lin K.J., Liu C.L., Gear C.W. (1992); "Imprecise Computation"; IOS Press; Washington D.C. , 1992, pg. 160-69

[LNL87] Lin K.J., Natarajan S., Liu J.W.-S. (1987); "Imprecise Results: Utilizing Partial Computations in Real-Time Systems"; *Proceedings of the 8th IEEE Real-Time Systems Symposium*; December 1987

[LSS87] Lehoczky J.P., Sha L., Strosnider J. (1987); "Enhancing Apreiodic Responsiveness in a Hrad eal-Time Environment"; Proceedings of IEEE Real-Time Systems Symposium; Los Alamitos. CA , 1987

[MCG98] Mok A.K., Chen D., Guangtian L. (1998); "Semantics and Resource Management of Real-Time Database Systems"; (to be published)

[MoL77] Morgan H.L., Levin K.D. (1977); "Optimal Program and Data Location in Computer Networks"; *Communications of the ACM* Vol. 20, No. 5 (1977)

[MoL97] Mok A., Liu G. (1997); "Early Detection of Timing Constraint Violations at Runtime"; Proceedings of the 18th *IEEE Real-Time Systems Symposiums;* San Francisco, California; December 1997

[Mur83] Murthy K. (1983); "An Aproximation the the File Allocation Problem in Computer Networks"; *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on principles of database systems* (1983)

[NaS94] Natale M.D., Satnkovic J.A. (1994); "Dynamic End-to-End Guarantees in Distributed Real-Time Systems"; *IEEE Real-Time Systems Symposiums;* December 1994

[NoA87] Noe J.D., Anressian A. (1987); "Effectiveness of Replication in Distributed Computer Networks"; *Proceedings of the 7th International IEEE Conference on Distributed Computing Systems*, West Berlin, September 1987

[OhM96] Oh S.K., MacEwen G.H. (1996); "Task Behavior Monitoring for Adaptive Real-Time Communication"; *Real-Time Systems* Vol. 11 No. 2.

[Pol94] Poledna S. (1994); "Replica Determinism in Distributed Real-Time Systems: A Brief Survery"; *Real Time Systems* Vol. 6 No. 3 (1994)

[Pu86] Pu C. (1986); "Replication of Nested Transactions in the EDEN Distributed System"; *Ph.D. dissertation*, University of Washington

[Raj89] Rajkumar R. (1989); "Task Synchronization in Real-Time Systems"; *Ph.D. Dissertation,*, Carnegie Mellon University (1989)

[Rag93] Rago S.A. (1993); *UNIX System V Network Programming*; Addison-Wesley, 1993

[RaP95] Ramamritham K., Pu C.(1995); "A Formal Characterization of Epsilon Serializability"; *IEEE*

Transactions on Knowledge and Data Engineering; **Vol. 7**, **No. 6**, December 1995

[RoS96] Rosu D.I., Schwan K. (1996); "Improving Protocol Performance by Dynamic Control of Communicating Resources"; *2nd IEEE International Conference on Engineering Complex Computer Systems;* 1996

[RRJ92] Raju S.C., Rajkumar R., Jahanian F. (1992); "Monitoring Timing Constraints in Distributed Real-Time Systems"; Proceedings of the 13th *IEEE Real-Time Systems Symposiums;* December 1992

[RSL89] Rajkumar R., Sha L., Lohoczky J.P. (1989); "An Experimental Investigation of Synchronization Protocols"; *Proceedings of the 6th IEEE Workshop on Real-Time Operating Systems and Software*; Pittsburgh 1989

[Sch93] Werner S. (1993); "The Testability of Distributed Real-Time Systems"; Kluwer Academic Publishers, 1993

[Seg76] Segall A. (1976); "Dynamic File Assignment in a Computer Network"; *IEEE Transaction on Automatic Control;* Vol. 21, No. 2 (1976)

[Seg97] Segbert G. (1997); "Entwurf und Implementierung eines Run-Time Monitors für das verteilte, sicherheitskritische Betriebssystem Melody"; *Masters Thesis*; Universität Dortmund

[SeS79] Segall A., Sanell N.R. (1979); "Dynamic File Assignment in a Computer Network - part II: Decentralized Control"; *IEEE Transaction on Automatic Control;* Vol. 24, No. 5 (1979)

[ShG92] Sha L., Goodenough J. (1992); "Real-Time Scheduling Theory and Ada"; *Mission-Critical Operating Systems*; IOS Press 1992; Washington D.C., 1992, pg. 294-319

[SLR86] Sha L., Lehoczky J.P., Rajkumar R. (1986); "Solutions for Some Practical Problems in Real-Time Scheduling"; *Proceedings of IEEE Real-Time Systems Symposium*; Los Alamitos. CA , 1986

[SLS88] Sprunt B., Lehoczky J.P., Rajkumar R. (1988); "Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm"; *Proceedings of the 9th IEEE Real-Time Systems Symposium*; Huntsville, Alabama 1988

[Smi81] Smith A.J. (1981); "Long Term Migration: Development and Evaluation of Algorithms"; *Communications of the ACM;* Vol. 24, No. 8 (1981)

[SpB96] Spuri M., Buttazzo G. (1996); "Scheduling Aperiodic Tasks in Dynamic Priority Systems"; *Real-Time Systems Journal*; Vol. 10 No. 2

[SSL89] Sprunt B., Sha L., Lehoczky J.P. (1989); "Aperiodic Task Scheduling for Hard Real-Time Systems"; *Real-Time Systems Journal*; Vol. 1 No. 1

[Sta96] Stange C. (1996); "Adaptives File Assignment in verteilten, sicherheitskritischen Betriebssystemen"; *Masters Thesis*; Universität Dortmund

[Ste92] Stevens R.W. (1992); *Programmieren von UNIX-Netzen*; Hanser, München, Prentice-Hall, London, 1992

[Ste94a] Stevens R.W.(1994); *TCP/IP IIllistrated, Volume 1 The Protocols*; Addison-Wesley, 1994

[Ste94b] Stevens R.W.(1994); *TCP/IP IIllistrated, Volume 2 The Implementation*; Addison-Wesley, 1994

[Tan93] Taneja S.K. (1993); "Distributed Computing in Computer--Integrated Manufacturing Systems"; *Ph.D.Dissertation*; Wayne State University, July, 1993

[TFC90] Tsai J.P., Fang K.Y., Chen H.Y. (1990); "A Noninvasive Architechture to Monitor Real-Time Distributed Systems"; *IEEE Computer;* Vol. 23, No. 3 (1990)

[TiB95] Davis R., Tindell K.W., Burns A. (1995); "Flexible Scheduling for Adaptable Real-Time Systems"; *IEEE Computer;* 1995

[Tin94] Tindell K. (1994); "Allocating Real-Time Tasks"; *Journal of Real-Time Systems;* Vol 4, No. 2(1994)

[TKM88] Tokuda H., Kotera M., Mercer C.W. (1988); "A Real-Time Monitor for a Distributed Real-Time Operating Systems"; *Proceedings of the ACM Workshop on Parallel and Distributed Debugging;* 1988

[WaH95] Wannemacher M., Halang W.A. (1995); "GPS-Bierte Zeitgeber: Real-zeitsysteme werden endlich echtzeitfähig"; *Proceedings PEARL94-Workshop über Realzeitsysteme*; P. Holleczek (Hrsg.), Reihe Informatik aktuell, Berlin-Heidelberg-New York: Springer 1995

[WeA89a] Wedde H.F., Alijani G.S., Baran D., Kang G., Kim B.K.(1989); "Adaptive Real-Time File Handling in Local Area Networks"; *Proceedings of the EUROMICRO '89 Workshop on Real-Time Systems*, Como/Italy, June 1989

[WeA89b] Wedde H.F., Alijani G.S., Huizinga D.B., Kang G., Kim B.K. (1989); "Real time file performance of a completely decentralized adaptive file system"; *Proceedings of the Int. IEEE Symposium on Real-Time Systems*, Santa Monica, CA, December 1989

[WeA89c] Wedde H.F., Alijani G.S., Huizinga D., Kang G., Kim B.(1989); "DRAGON SLAYER/MELODY: Distributed Operating System Support for Mission-Critical Computing"; *Proeedings. of the 1989 Workshop on Operating Systems for Mission-Critical Computing*; University of Maryland, College Park, September 1989

[WeA90] Wedde H.F., Huizinga D., Kang G., Kim B.(1990); "MELODY: A Complete Decentralized Adaptive File System for Handling Real-Time Tasks in Unpredictable Environments"; *Real-Time Systems* Vol. 2 No. 4

[WeD91a] Wedde H.F., Daniels D.C., Huizinga D.M. (1991); "Efficient Distributed Resource Scheduling for Adaptive Real-Time Operation Support"; *Springer Lecture Notes in Computer Science*; Vol. 497 (1991)

[WeD91b] Wedde H.F., Daniels D.C. (1991); "Distributed Resource Scheduling under Real-Time Constraints"; *Second Great Lakes Computer Science Conference*, Kalamazoo, Michigan, October 1991

[WeD94] Wedde H.F., Dekker M. (1994); "Real-Time Operating Systems and Software: State of the Art and Future Challenges, in"; A. Kent, J. Williams: *Encyclopedia of Microcomputers* Vol. 14 (1994)

[WeK93] Wedde H.F., Korel B., Lind J.A. (1993); "Highly Integrated Task and Resource Scheduling for Mission-Critical Systems"; *Proceedings of the EUROMICRO'93 Workshop on Real-Time Systems*, Oulu, Finland, June 1993

[WeL94] Wedde H.F., Lind J.A.,Eiss A. (1994); "Achieving Dependability in Safety-critical Operating Systems Through Adaptability and Large-Scale Functional Integration"; *Proceedings of the ICPAPDS'94 International Conference on Parallel and Distributed Systems*; Hsinchu, Taiwan, December 1994

[WeL95] Wedde H.F., Lind J.A., Eiss A.(1995); "Incremental Experimentation: A Methodology for Designing and Analysing Distributed Safety-Critical Systems"; *Proceedings of the EUROMICRO '95 Workshop on Real-Time Systems*, Odense, Denmark, June 1995

[WeL97] Wedde H.F., Lind J.A. (1997); "Building Large, Complex, Distributed Safety--Critical Systems"; *Real-Time Systems*, Vol. 13, No. 3

[WeL98] Wedde H.F., Lind J.A. (1998); "Integration of Task Scheduling and File Services in the Safety-Critical System MELODY"; *Proceedings of the EUROMICRO '98 Workshop on Real-Time Systems*, Berlin, Germany, 1998

[WeS96] Wedde H.F., Stange C., Lind J.A. (1996); "Integration of Adaptive File Assignment into Distributed Safety-Critical Systems"; *WRTP'96, 21st IFAC/IFIP Workshop on REAL TIME PROGRAMMING*, Gramado, RS, Brazil, November 1996

[WeX92] Wedde H.F., Xu M.(1992); "Scheduling Critical and Sensitive Tasks with Remote Requests in Mission-Critical Systems"; *Proceedings of the EUROMICRO '92 Workshop on Real-Time Systems*; Athens, Greece, June 1992

[XRH98] Xiong M., Ramamritham K., Haritsa J., Stankovic J.A. (1998); "Regulating Concurrent Accesses to Replicated Data in Distributed Real-Time Databases"; *Proceedings of the 19th IEEE Real-Time Systems Symposiums;* December 1998 (to be published)

[XSS98] Xiong M., Sivasankaran R., Stankovic J.A., Ramamritham K., Towsley D. (1998); "Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics"; *Proceedings of the 17th IEEE Real-Time Systems Symposiums;* December 1996

[XuP93] Xu J., Parnas D.L., "On Satisfying Timing Constraints in Hard Real-Time Systems"; *IEEE Transaction on Software Engineering*; Vol. 19 No. 1 (1993)

[Yu85] Yu C.T. (1985); "Adaptive File Allocation in Star Computer Network"; *IEEE Transaction on Software Engineering;* Vol. 11, No. 9 (1985)

[ZSA92] Zhou H., Scwan K., Akyildiz I. (1992); "Performance Effects of Information Sharing in a Distributed Multiprocessor Real-Time System"; *Real-Time Systems Symposium;* December 1992