Efficiently Tolerating Failures in Asynchronous Real-Time Distributed Systems ⋆

Peng Li *

Real-Time Systems Laboratory
Department of ECE, Virginia Polytechnic Institute and State University
Blacksburg, VA 24061, USA
Phone: 1-540-231-1418, FAX : 1-540-231-3362

Binoy Ravindran

Real-Time Systems Laboratory
Department of ECE, Virginia Polytechnic Institute and State University
Blacksburg, VA 24061, USA
Phone: 1-540-231-3777, FAX : 1-540-231-3362

Abstract

We present a proactive resource allocation algorithm, called BEA, for fault-tolerant asynchronous real-time distributed systems. BEA considers an application model where trans-node application timeliness requirements are expressed using benefit functions, and anticipated workload during future time intervals are expressed using adaptation functions. Furthermore, BEA considers an adaptation model where subtasks of application tasks are replicated at run-time for tolerating failures as well as for sharing workload increases. Given such models, the objective of the algorithm is to maximize the aggregate real-time benefit and the ability to tolerate host failures during the time window of adaptation functions. Since determining the optimal solution is computationally intractable, BEA heuristically computes near-optimal resource allocations in polynomial-time. We show that BEA can achieve almost the same fault-tolerance ability as full replication, and accrue most of real-time benefit that full replication can accrue. In the meanwhile, BEA requires much fewer replicas than full replication, and hence is cost effective.

Key words: real-time parallel and distributed systems, resource allocation and management, fault tolerance, real-time distributed algorithms

1 Introduction

Asynchronous real-time distributed systems are emerging in many domains including defense, telecommunication, and industrial automation for the purpose of strategic mission management (Jensen and Ravindran, 2002). Such systems are fundamentally distinguished by the significant run-time uncertainties that are inherent in their application environment and system resource states (Jensen and Ravindran, 2002; Koob, 1996; Jensen, 1992). Consequently, it is difficult to postulate upper bounds on application workloads for such systems that will always be respected at run-time.

To deal with such non-determinism, recent advances in real-time and distributed systems research (DARPA, 1997) have produced quality of service (QoS) technologies that allow applications to specify and negotiate real-time requirements. The real-time QoS techniques consider application models where applications can operate at multiple, discrete "levels" of service. A level is a strategy for doing an application's work and is characterized by a resource usage such as CPU and network utilization, a QoS dimension such as timeliness of computations, and a user-specified benefit. Thus, if all timing requirements of all applications cannot be satisfied at run-time (due to the workload hypothesis being violated at run-time), then an adaptation mechanism can determine the "right level" of QoS to optimize a system-wide criteria such as maximizing aggregate benefit. Examples of such real-time QoS techniques are presented in (Abdelzaher and Shin, 1998; Brandt et al., 1998; Lee, 1999; Ravindran, 2002; Rosu et al., 1997) and have been implemented as real-time middleware (Bettati, 1997).

However, most of the real-time QoS techniques studied in the literature are "reactive" in the sense that the techniques dynamically react to workload fluctuations and system failures as and when they occur. Thus, when workload fluctuations and failures occur, the techniques detect such conditions, and reallocate resources so that a system-level criteria is optimized. We believe that such reactive techniques may not *always* be effective, due to their significant run-time overheads, especially when workload fluctuations and failure occurrences are highly "bursty." In fact, due to the high cost of computing optimal or near-optimal resource allocation decisions, many reactive resource allocation techniques such as (Ravindran, 2002; Rosu et al., 1997) sacrifice the optimality of their decisions for the speed with which the decisions can be computed.

To overcome this difficulty, in this paper, we present a proactive resource al-

---

⋆ This work was supported by the U.S. Office of Naval Research under Grant N00014-99-1-0158 and N00014-00-1-0549.
∗ Corresponding author
Email addresses: peli2@vt.edu (Peng Li), binoy@vt.edu (Binoy Ravindran).

location algorithm called Best Effort Allocation (or BEA), for fault-tolerant asynchronous real-time distributed systems. BEA is proactive in the sense that it enables user-triggered, user-specified, application-specific, and situation-specific resource allocation.

BEA considers an application model where trans-node application tasks have end-to-end timeliness that are expressed using benefit functions. A benefit function describes a task's benefit to the system as a function of its completion time (Jensen, 1992). For example, a simple deadline timing constraint can be expressed as a step benefit function. To facilitate proactive QoS allocation, BEA uses *adaptation functions*—a concept that we originally developed in (Hegazy and Ravindran, 2002). An adaptation function is specified by the user and describes the user-anticipated application workload during future time intervals. Furthermore, our failure model assumes that there are several *logical* host groups in the system. Processors within the same host group have the same reliability and have high possibility of failing simultaneously, i.e., simultaneous fail-stops. Example of a host group is a combat zone in a ship, which may be destroyed by enemy attack simultaneously. Thus, only replication across host groups can increase the fault-tolerance ability of a subtask.

BEA considers an adaptation model where subtasks of application tasks can be transparently and automatically replicated at run-time for tolerating end-host failures, besides for sharing workload increases. More specifically, we assume that two types of subtasks, namely "worker subtasks" and "manager subtasks" are interleaved within an end-to-end task. A worker subtask can be replicated at run-time and each replica can share workload. On the other hand, replicas of a manager subtask do not share workload; they form a "process group" (Birman, 1993) for tolerating end-host failures.

Furthermore, the algorithm targets a system model where each host is connected through an Ethernet switch. In the meanwhile, hosts in the system use best-effort real-time scheduling algorithms such as DASA (Clark, 1990) for scheduling and resource access control.

Given such application, adaptation, failure, and system models, the objective of BEA is to maximize the aggregate real-time benefit and fault-tolerance ability of the application during future time intervals described by the adaptation. This problem can be shown to be $\mathcal{NP}$-hard. Thus, the BEA algorithm we present in this paper is a heuristic algorithm that produces sub-optimal resource allocations in polynomial-time.

To study how well BEA performs, we compare the performance of BEA with full replication strategy. Experimental results show that BEA can achieve almost the same fault-tolerance ability as full replication, and accrue most of the real-time benefit that full replication can. However, BEA requires much

fewer replicas than full replication and hence is cost effective.

Thus, the major contribution of the paper is the polynomial-time BEA algorithm that seeks to maximize aggregate real-time *and* fault-tolerance ability of asynchronous real-time distributed systems.

The rest of the paper is organized as follows: Section 2 describes the application, adaptation, failure, and system models considered in this work. Section 3 informally states the resource allocation problem that we are addressing in this paper. Section 4 presents the BEA algorithm. We show the computational complexity of BEA in Section 5. In Section 6, we discuss the experimental evaluation of the algorithm. Finally, the paper concludes with a summary of the work and its contributions in Section 7.

2 Application, Adaptation, System, and Failure Models

2.1 The Application Model

We denote the set of tasks in the application by the set $T = \{T_1, T_2, T_3 \ldots\}$, where a task can be either periodic or aperiodic. Each aperiodic task T_j has a "triggering" periodic task T_k that triggers its execution. After a periodic task T_k completes its execution, it may trigger the execution of the corresponding aperiodic task T_j.

Our task model is inspired by typical applications in defense domain such as US Navy's Anti-Air Warfare (AAW) system (Welch et al., 1998). An AAW system usually has inputs from radar(s) (known as "radar tracks"), which periodically swipe a certain air space. Furthermore, input radar tracks are processed to identify potential threats. In case that threats are identified, an aperiodic task may be triggered to engage weapons to destroy the threats. To better understand AAW, a software prototype that approximated AAW called *DynBench* (Shirazi et al., 2000), was developed as part of DARPA's Quorum program (DARPA, 1997).

The period of a periodic task T_i is denoted as $period(T_i)$. If T_i is aperiodic, $period(T_i)$ denotes the period of the periodic task that triggers task T_i.

Each task T_i is assumed to consist of a set of subtasks (executable programs), which execute "serially." We use the notation $T_i = [st^i_1, m^i_1, st^i_2, m^i_2, \ldots, st^i_n, m^i_n]$ to represent a task T_i that consists of n subtasks and n messages to be executed and transmitted in series, such that subtask st^i_j $(1 < j \leqslant n)$ cannot execute before message m^i_{j-1} arrives. For convenience, we denote the set of subtasks

of a task T_i as $ST(T_i) = \{st^i_1, st^i_2, st^i_3, \ldots, st^i_n\}$ and the set of inter-subtask messages of a task T_i as $MS(T_i) = \{m^i_1, m^i_2, m^i_3, m^i_4, \ldots, m^i_n\}$.

2.1.1 Timing Requirements

We use Jensen's benefit functions (Jensen, 1992) for expressing application timeliness requirements. Note that a benefit function measures a task's benefit [1] to the system as a function of its completion time. Furthermore, we assume "step" benefit functions for all tasks such as the one shown in Figure 1. Thus, completing a task anytime before its deadline will result in uniform benefit; completing it after the deadline will result in zero benefit. We denote the height of the benefit function of a task T_i as $maxRTB_i$.

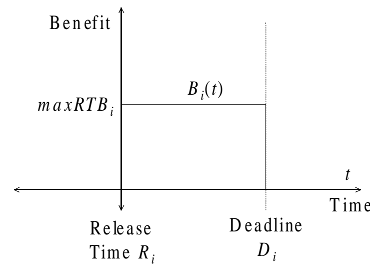

Fig. 1. A Task Benefit Function

2.1.2 Application Workloads

We model the workload of a subtask and that of an inter-subtask message as the number of data objects that they need to process and transmit, respectively. The motivation for this model is due to the fact that the number of sensor reports and aperiodic events (or data objects) processed and transmitted by subtasks and messages, respectively, constitute the most significant part of the application workload in many real-time distributed applications that we regard as asynchronous (Clark et al., 1999; Welch et al., 1998).

Furthermore, the major element of uncertainty in the processing and communication latencies in such systems is due to the uncertainty in the number of data objects that the application has to process, as they are dependent upon the applications' external environment. Thus, we define the end-to-end execution latency of a task as the time that the task takes to complete the processing of a single data object.

We assume that application-profile functions that can estimate subtask execution times as a function of data objects are available. We regard such execution time estimates as worst-case lower bounds for processing a *single* data object. The profile functions can be determined by application profiling and measurement. Note that we assume that application subtask-profile func-

[1] "Benefit" is also called "value" or "utility" in real-time literatures.

tions are available, but the parameters of the functions—*the number of data objects*—are unknown.

2.2 Adaptation Model

We consider an adaptation model for the application where some subtasks of application tasks called "worker subtasks," can be *replicated* at run-time. The idea behind replication of subtasks is that once a subtask is replicated, replicas of the subtask can share the workload that was processed by the original subtask. Therefore, the end-to-end task latency can be reduced. Thus, replication is allowed as a means to reduce task latencies when task workloads increase at run-time. Besides reducing end-to-end latency, we view replication is the key to fault-tolerance. In general, the more replicas the higher fault-tolerance ability a task has.

Apart from the worker subtasks, we assume that all other subtasks of a task are non-replicable. We call such subtasks "manager subtasks." Manager subtasks are assumed to serve as points of synchronization and of workload distribution. Since we are assuming a "serial" task structure (as discussed in Section 2.1), the manager subtasks and worker subtasks become interleaved in the task structure. Thus, a manager subtask buffers the incoming messages from replicas of the predecessor worker subtask until messages from all replicas are received. The workload is then distributed among replicas of the successor worker subtask. To tolerate single-point failures, there may be several identical copies of a manager subtask simultaneously executing on different processors. Notice that these copies do not share workload. Instead, we assume that they form a "process group" (Birman, 1993), where group membership services such as leader election and failure detection are available.

For simplicity in the design of the application and the resulting application model, we assume that the workload of a subtask is equally distributed among all its replicas.

The state consistency of the subtask replicas is not considered in this work, as we assume that the tasks process data objects that are "continuous" in the sense that their values are obtained directly from a sensor in the application environment, or computed from values of other such objects. The replicas are thus assumed to be *temporally consistent* without applying every change in value, due to the continuity of physical phenomena (Jensen, 1992).

We use adaptation functions, a concept that we originally developed in (Hegazy and Ravindran, 2002), for expressing anticipated workload scenarios of the application during future time intervals. The adaptation functions describe the anticipated application workload as a function of the time (or a reference point

such as task period) at which it is anticipated to occur. The functions can have arbitrary shapes and are user-specified for each task. We regard the origin of the functions' axes as the start time of the expected scenario for the workload. The function is specified for a fixed duration of time into the future, and ends at a time instant called the "time horizon."

In this paper, we use task periods as reference points for adaptation functions. For a periodic task, the anticipated workload during period p is defined as the number of data objects that are anticipated during p. For an aperiodic task, the anticipated workload during period p is defined as the number of events that the triggering periodic task (of the aperiodic task) is anticipated to produce at the end of its period p. Furthermore, we assume that the anticipated workload is a constant during the task period, but may be different for different periods. An example adaptation function is shown in Figure 2, where G_i is the start time for specifying the anticipated workload and H_i is the time horizon.

The anticipated workload scenarios—and thus the adaptation functions—can be dynamically modified as and when the user's perception regarding the future workload changes. Specification of the functions and the enactment of the functions (as and when desired) will trigger proactive resource allocation.

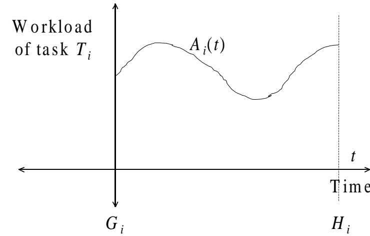

Fig. 2. A Task Adaptation Function

Thus, adaptation functions facilitate a "human-in-the-loop" approach, where adaptation of the application can be performed according to user's perception of the future operational situation of the application, thereby facilitating user-desired and situation-specific adaptation of the system.

It is important to notice that the proactive resource allocation strategy that we propose is not a "stand-alone" mechanism for adaptive resource allocation. Rather, it complements reactive resource allocation techniques.

2.3 The Failure Model

We assume a fail-stop model for the end-hosts, i.e., a host simply halts when it fails. This is because of our target application domain—defense domain—where most end-host failures are due to hostile environment, such as missile attacks. For specifying reliability functions, we consider a *logical* partition $\{H_i\}_{i\in\{1,..,g\}}$ of the host set H into g groups such that $\forall i, H_i \neq \phi$, $\bigcup_{i=1}^{g} H_i = H$, and $\bigcap_{i=1}^{g} H_i = \phi$. Host-group partitions are assumed to be user-specified according to the application-specific requirements.

We consider a "forward recovery" model for achieving fault-tolerance. Application subtasks are dynamically replicated and maintained in different host-group partitions for achieving fault-tolerance. When a replica fails due to an end-host failure, the failure is tolerated by ensuring that additional replicas exist in other end-host group partitions that can process the subtask workload in successive task periods. Thus, the fault-tolerance mechanism is a forward recovery mechanism in the sense that it provides continued availability of task functionality. Our rationale for considering such a fault-tolerance model is our application and adaptation models, where tasks process continuous data objects. The replicas are temporally consistent without applying every change in value, due to the continuity of physical phenomena. For example, a lost radar track (for an air craft) can be recovered during the subsequential radar swipes.

We define the "*importance level*" of a task T_i as I_i, which is also user-defined. The importance level provides a sense of fault-tolerance requirement for a task, and hence is a orthogonal QoS dimension to the timing constraint. Notice that a high real-time benefit task may not have high importance level, and vice versa.

2.4 The System Model

We consider a switched Local Area Network segment, where each end-host is connected to a unique port of an Ethernet switch through a full-duplex link. We denote the set of end-host processors by the set $PR = \{p_1, p_2, p_3, \ldots, p_m\}$. We assume that the clocks of the processors are synchronized using a protocol such as (Mills, 1995).

For process scheduling and resource access control, we consider the DASA best-effort real-time scheduling algorithm (Clark, 1990) that explicitly incorporates benefit functions for scheduling. The DASA algorithm employs the notion of "value-density," which measures the amount of "value" (or benefit as in this paper) that can be accrued per unit time. More specifically, the DASA algorithm first sorts all ready tasks according to their value densities. Tasks are then inserted into a tentative schedule in decreasing order of their value densities. Note that to mimic the output of an EDF scheduler, the tentative schedule is deadline-ordered. If insertion of a task T_i causes deadline miss of any of the existing tasks in the tentative schedule, T_i is removed from the tentative schedule. Otherwise, T_i remains in the tentative schedule. This procedure is repeated until all ready tasks have been examined.

Our choice of DASA is due to the fact that it outperforms EDF during overloaded situations, and performs the same as EDF during under-loaded situations where EDF is optimal. Furthermore, implementations of DASA are avail-

able in operating systems including the MK7.3 kernel (MK7.3a, 1998) and our own Linux implementation (Li et al., 2001). Notice that we use non-preemptive version of DASA whenever proper, i.e., for physical frame scheduling on switch ports and host network interfaces.

Besides the switched network for transmitting data message, we assume that each host machine is also attached to another local area network for transmitting control messages, such as the messages used in the process group membership services. This dual network architecture is primarily for high performance, as well as for tolerating network failure, and has been adopted in the U.S. Navy HiPerD testbed (HiPerD, 1997). Thus, the communication of data messages is not affected by control message traffic.

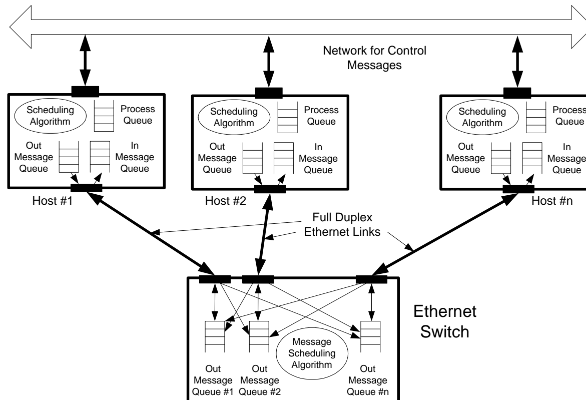

Fig. 3. The System Model

3 Resource Allocation Objective

Given the application, adaptation, failure, and system models described in Section 2, our objective is to maximize the aggregate real-time benefit *and* fault-tolerance ability during the time window of task adaptation functions. Ideally, we desire all tasks meet their deadlines and every subtask of a task has replicas within each host groups. However, this objective is subject to the total available resource constraint.

Thus, our QoS resource allocation objective can be informally stated as follows: *"Given the application, adaptation, failure, and system models described in Section 2, what allocation—number of replicas for each subtask and their end-hosts—will maximize the aggregate real-time benefit and fault-tolerance ability during the time window of the task adaptation functions?"*

This problem can be shown to be $\mathcal{NP}$-hard as it subsumes the multiple 0-1 knapsack problem.[2] Thus, the BEA algorithm is a heuristic algorithm that solves this problem in polynomial-time, but not necessarily produce optimal

[2] For brevity, we omit the proof here. However, the proof is similar to that presented in (Hegazy, 2001) for a different resource allocation problem.

allocations.

4 The BEA Algorithm: Heuristics and Rationale

Observe that the DASA scheduler will abort any task that misses its deadline. Under such a situation, fault-tolerance cannot be achieved even if the task has several replicas. Thus, meeting task deadline is the prerequisite of fault-tolerance for the models used in this paper.

Therefore, the BEA algorithm allocates resources in two stages: stage I allocates resources to satisfy task deadlines if possible, and stage II may allocate more resources to improve the task fault-tolerance ability. Furthermore, the desired properties of BEA include:

(1) *During stage I, allocate resources in decreasing order of $maxRTB_i$.* Recall that $maxRTB_i$ is the height of a step benefit function, as defined in Section 2.1.1. By doing so, we increase the possibility of maximizing the aggregate real-time benefit. This is because, the task selected next for resource allocation is always the one with the highest real-time benefit among the unallocated tasks;
(2) *During stage I, proceed to the next task adaptation period for resource allocation, if the current allocation already satisfies the task deadline, or even full replication cannot satisfy the task deadline.* By doing so, we save system resources, which can be allocated to tasks with lower values of $maxRTB_i$. This will increase the possibility of obtaining contributions of non-zero real-time benefit and fault-tolerance from such tasks;
(3) *During stage II, allocate resources in decreasing order of task importance levels that are defined in Section 2.3.* By doing so, the algorithm seeks to maximize the fault-tolerance ability of important tasks as many as possible.
(4) *Decompose task-level resource allocation problem into subtask-level resource allocation problems, i.e., allocating resources to subtasks of tasks.* The rationale behind this heuristic is that solving the task-level resource allocation problem will require a holistic analysis of the system. This can be computationally expensive. Therefore, by decomposing the task-level problem into subtask-level sub-problems and solving the sub-problems, we seek to reduce the overhead of computing a near-optimal solution. Since we are focusing on step-benefit functions for tasks, the decomposition can be done by assigning deadlines to subtasks and messages of a task from the task deadline in such a way that if all subtasks and messages of the task can meet their respective deadlines, then the task will be able to meet its deadline. Using this heuristic, we can now determine the replica needs of a task by determining the replica needs of subtasks

of the task.

Thus, BEA allocates resources using the heuristics presented here. To assign deadlines to subtasks and messages of a task from the end-to-end task deadline, BEA uses a variant of the equal flexibility (EQF) algorithm presented in (Kao and Garcia-Molina, 1997). The EQF algorithm decomposes a task end-to-end deadline into subtask deadlines such that slacks of subtasks are proportional to their execution times. Observe that subtask execution times are dependent upon not only anticipated workload, but also the number of replicas, which bears the essential part of the adaptation mechanism we proposed here. As suggested in (Kao and Garcia-Molina, 1997), once a worker subtask is replicated, its deadline is reduced accordingly. Furthermore, since manager subtasks cannot be replicated to reduce execution times, they are assigned the longest possible deadlines. This long deadline assignment strategy decreases the possibility of timing failures of the manager subtasks, which in turn results in timing failure of the whole task. Therefore, for each period of the adaptation function, BEA computes the end-to-end task slack by assuming all worker subtasks within the task are fully replicated. Then, this end-to-end slack is assigned to subtasks and messages proportional to their execution times and communication delays, respectively.

Since we are considering the DASA scheduling algorithm, which uses benefit values of subtasks for computing scheduling decisions, the user-specified real-time benefit of a task must be mapped into benefit values for subtasks of the task. Therefore, BEA defines the real-time benefit of a subtask as simply the real-time benefit $maxRTB_i$ of its parent task.

Algorithm 1 High level description of BEA

1: **Input:** a set of tasks $T = \{T_1, T_2, \ldots, T_n\}$ and an adaptation window W;
2: `/* Stage I: allocate resources to satisfy task deadlines */`
3: **for** each task $T_i \in T$ in descending order of $maxRTB_i$ **do**
4:    **for** each period $j = 1$ to $\lceil \frac{W}{period(T_i)} \rceil$ **do**
5:       decompose the task deadline to subtask and message deadlines by EQF;
6:       **for** each subtask $k = 1$ to $|ST(T_i)|$ **do**
7:          $DL_Allocation\left(st_k^i, j, Adapt\left(st_k^i, j\right)\right)$;
8: `/* Stage II: allocate resources to improve fault-tolerance ability */`
9: **for** each task $T_i \in T$ in descending order of importance levels I_i **do**
10:    **for** each period $j = 1$ to $\lceil \frac{W}{period(T_i)} \rceil$ **do**
11:       **for** each subtask $k = 1$ to $|ST(T_i)|$ **do**
12:          $FT_Allocation\left(st_k^i, j, Adapt\left(st_k^i, j\right)\right)$;

Algorithm 1 shows the high-level pseudo-code of the BEA algorithm. BEA first allocates resources in decreasing order of $maxRTB_i$ values such that task deadlines can be satisfied, using algorithm *DL_Allocation*. The algorithm checks whether each subtask of a task within each period can meet their in-

dividual deadlines by detecting overload situations. If BEA finds a manager subtask will miss its deadline, allocation for the entire task during that period is aborted. Accordingly, all resources for that period is de-allocated. This is because the deadline assignment strategy already assigns the longest possible deadline to the manager subtask. Thus, replication of worker subtasks cannot increase the manager subtask deadline, and hence cannot help to meet its deadline. However, if BEA determines a worker subtask st^i_j will miss its deadline, the algorithm replicates st^i_j until either st^i_j can meet its deadline or no underloaded processor is found. In the case of no underloaded processor, the allocation process is also aborted and resources for that period are de-allocated. Description of the *DL_Allocation* algorithm is presented in Algorithm 2.

Once stage I allocation is completed, BEA uses algorithm *FT_Allocation* to maximize the task fault-tolerance ability. Recall that hosts within the same host group are likely to fail simultaneously. Thus, replicas within the same host group cannot improve the fault-tolerance ability. For each task, the presence of one or more replicas within each host group will maximize its fault-tolerance ability. Therefore, BEA examines whether each subtask of a task has at least one replica within each group. If subtask st^i_j does not have a replica within host group H_g, BEA tries to allocate a replica of st^i_j on processor $p \in H_g$. It is important to notice that allocation of the new replica must not cause deadline miss of any of the existing replicas.

Since the anticipated workload may be different for different task periods in the adaptation time window, BEA allocates replicas for each period in the time window, starting from the most recent period and proceeding to the least recent. Furthermore, for each task T_i, for each period, the algorithm allocates replicas for subtasks of the task starting from subtask st^i_1 and proceeding to subtask st^i_m.

We now discuss how BEA (1) analyzes processor overloads to select end-hosts for subtask replicas in stage I, and (2) how the algorithm selects a processor $p \in H_g$, if no replica is present in H_g. These steps are discussed in the subsections that follow.

4.1 Analyzing Processor Overloads

BEA analyzes end-host processor overloads to select an end-host processor for a worker subtask replica. The rationale behind overload analysis is that if an end-host processor is under-loaded after being assigned a subtask with a given workload, then clearly, the subtask must be able to complete its execution by its deadline as the DASA algorithm is equivalent to EDF during under-loaded situations. Thus, if an end-host processor is under-loaded, we can conclude

Algorithm 2 The *DL_Allocation* algorithm

1: **Input:** subtask st_k^i, its period number j, and workload specified by $Adapt\left(st_k^i, j\right)$;
2: **if** st_k^i is a manager subtask **then**
3:   **if** st_k^i can meet its deadline **then**
4:     **return success**;
5:   **else**
6:     de-allocate all resources for period j of task T_i;
7:     **return failure**;
8: **else**
9:   **while** st_k^i cannot satisfy its deadline **do**
10:     replicate st_k^i using $AssignProcessor\left(st_k^i, j, Adapt\left(st_k^i, j\right)\right)$;
11:   **if** st_k^i can meet its deadline **then**
12:     **return success**;
13:   **else**
14:     de-allocate all resources for period j of task T_i;
15:     **return failure**;

that the processor is a "good" candidate for the subtask for the workload that the subtask has to process.

On the other hand, if a processor is overloaded after assigning the subtask to the processor, then the processor is not a good candidate as the allocation will cause one or more subtasks on the processor to miss their deadlines.

Pseudo-code of the *AssignProcessor* algorithm that selects a processor for a subtask replica is shown in Algorithm 3. Observe that *AssignProcessor* is called by *DL_Allocation* algorithm(Step 10) to determine the end-host processor of a subtask replica during the tentative resource allocation process.

Algorithm 3 The *AssignProcessor* algorithm

1: **Input:** subtask st_k^i, its period number j, and workload specified by $Adapt\left(st_k^i, j\right)$;
2: **while** underloaded processor for st_k^i has not been found **do**
3:   select the next unassigned processor q;
4:   **if** $OverloadCheck\left(st_k^i, q, Adapt\left(st_k^i, j\right) \Big/ \left(NumReplicas\left(st_k^i\right) + 1\right)\right) \neq$ **true then**
5:     assigns a replica of subtask st_k^i to processor q;
6:     increase the number of replicas of st_k^i by one;
7:     **return success;**
8: **return failure**;

To test for a processor overload, BEA constructs the schedule by assuming the EDF scheduler. Note that a task set is schedulable by EDF if and only if there is no overload. We formulate the core part of the overload analysis procedure in Algorithm 4. The algorithm accepts a list of subtask arrival events ordered by arrival times, and returns either "true" for overload situation or otherwise

"false". The schedule is constructed by making scheduling decisions at each scheduling event, which include subtask arrival and subtask termination. The events are computed using the subtask arrival times and the expected termination time of the currently running subtask ST_r. Note that ST_r is expected to terminate $RemainingTime\,(t)$ later than the "current" time t. If the earliest arrival time among subtasks in ST is later than the expected termination time of ST_r, then ST_r should terminate first. Otherwise, a new subtask will arrive at the processor while another subtask is executing on the processor. At each scheduling event, the algorithm always selects the earliest deadline subtask from ready queue(using $EarliestDL\,(RQ)$).

Algorithm 4 The core part of *OverloadCheck* algorithm

1: **Input:** n subtask arrivals $ST = \{ST_1, ST_2, \ldots, ST_n\}$, ordered by arrival times;
2: initialize ready queue $RQ = \phi$, running subtask $ST_r = \phi$, and its expected finishing time $\xi = \infty$;
3: **while** $ST \neq \phi$ **or** $RQ \neq \phi$ **do**
4:   select $ST_i \in ST$, which has the earliest arrival time among all tasks in ST;
5:   **if** $ST == \phi$ **or** $ArrTime\,(ST_i) \geq \xi$ **then**
6:     extract ST_r from RQ if any, $RQ = RQ - ST_r$; $t = \xi$;
7:     **if** $t > Deadline\,(ST_r)$ **then**
8:       **return true**;
9:   **else**
10:     extract ST_i from ST, $ST = ST - ST_i$;
11:     insert ST_i into RQ by deadline, $RQ = RQ \cup ST_i$; $t = ArrTime\,(ST_i)$;
12:   **if** $ST_r \neq \phi$ **then**
13:     update $RemainingTime\,(ST_r)$;
14:   make scheduling decision: $ST_r = EarliestDL\,(RQ)$;
15:   **if** $ST_r \neq \phi$ **then**
16:     $\xi = t + RemainingTime\,(ST_r)$;
17: **return false**;

Note that this overload test is true only under EDF. However, DASA is equivalent to EDF during under-load conditions and differs from EDF only during overload situations. Furthermore, the overload analysis can also determine the response time of a subtask if there is no overload, which is simply the interval between the arrival time of a subtask and its termination time. Thus,the overload analysis procedure can either determine an overload situation or otherwise determine the response time of a subtask replica that is being considered for execution on the processor.

To test for overload on an end-host processor, the arrival times of all *higher real-time benefit* subtasks on the processor must be known. For determining subtask arrival times, BEA assumes that a subtask completes its execution when all of its replicas complete their execution. This assumption is justified by the manager-worker task model presented in Section 2.1, where the manager subtask is a synchronization point. Therefore, the response time of a subtask is the longest response time among all its replicas. Further, the first subtask of

a task will arrive at the beginning of the period of its parent task; every other subtask will arrive after the elapse of an interval of time (since the beginning of the task period) that is equal to the sum of the message delays and subtask response times of all predecessor messages and all predecessor subtasks of the subtask, respectively. Furthermore, since stage I allocation is proceeded in descending order of task real-time benefit, higher real-time benefit subtask response times are known at the point of allocating resources for T_i.

Thus, the arrival time of a subtask can be determined as the sum of the response times of subtasks and communication delays of messages that precede the subtask (under consideration), and the arrival time of the parent task of the subtask. Given the arrival time of a task T_i, the arrival time of a subtask st_j^i of the task is therefore given by $ArrTime\left(st_j^i\right) = ArrTime\left(T_i\right) + \sum_{k=1}^{j-1}\left[ResponseTime\left(st_k^i\right) + Delay\left(m_k^i\right)\right]$, where $ArrTime\left(x\right)$ denotes the arrival time of a subtask or a task x, $ResponseTime\left(x\right)$ denotes the response time of a subtask x, and $Delay\left(x\right)$ denotes the communication delay of a message x.

As discussed previously, the response time of a subtask on a processor can be determined as part of the overload analysis. Similarly, BEA determines the message communication delays using the same overload analysis technique. However, since message scheduling is non-preemptive, the message scheduler does not need to be invoked at a message arrival event, unless the ready queue is empty. For brevity, we omit the communication delay analysis here.

4.2 Allocating resources for fault-tolerance

Algorithm 5 The *FT_Allocation* algorithm

```
1: Input: subtask st_k^i, its period number j, and workload specified by
   Adapt(st_k^i, j);
2: for each host group H_g do
3:   if st_k^i has no replica in H_g then
4:     StartTime = ArrTime(st_k^i, j);
5:     if st_k^i is a manager subtask then
6:       Duration = ExecutionTime(st_k^i, Adapt(st_k^i, j), 1);
7:     else
8:       Duration = ExecutionTime(st_k^i, Adapt(st_k^i, j), NumReplicas(st_k^i, j) + 1);
9:     for each processor p ∈ H_g do
10:      if p is idle during time interval [StartTime, StartTime + Duration]
         then
11:        assign one replica of st_k^i to p;
12:        break;
```

Stage I allocation of the BEA algorithm seeks to satisfy task deadlines. However, it is possible that some subtasks do not have replicas within each host

group, because their deadlines are satisfied by fewer replicas. By the failure model discussed in Section 2.3, the fault-tolerance ability of those subtasks and their parent end-to-end tasks are not maximized. Thus, the *FT_Allocation* algorithm examines the replica assignment of each subtask, from the most important task to the least important task, and checks whether their fault-tolerance ability is maximized. The idea is that the available resources after stage I allocation may not be sufficient to maximize the fault-tolerance ability of *all* tasks. Therefore, this important-task-first allocation heuristic increases the possibility of maximizing fault-tolerance ability of the most important tasks.

However, the new replicas for fault-tolerance should not cause deadline miss of any of the existing replicas. This goal could be expensive to achieve due to the complexity involved in detailed timing analysis. Moreover, the order of importance levels could be different from the order of task real-time benefits, which may contribute significant errors to timing analysis if recursion is not used. To avoid this expensive timing analysis, BEA adopts a straightforward approach: the algorithm tries to fit a new replica into a processor, which is idle during the execution time interval of that replica. We assume that fitting a replica into a spare time slot on processor p normally, though not necessarily, does not cause deadline miss of any of the existing replicas on all processors. Algorithm 5 shows the skeleton of the *FT_Allocation* algorithm.

5 Computational Complexity of BEA

For analyzing the complexity of BEA, we consider n tasks, p end-hosts, a maximum of m subtasks per task, a smallest task period of k, and a longest task adaptation window of length W.

OverloadCheck first computes the message communication delays to construct the subtask arrival list on a processor. In the worst case, there are $mn\lceil W/k\rceil$ messages arriving at the same output port of the switch. Ordering these messages by arrival times costs $O\left(mn\lceil W/k\rceil lg\left(mn\lceil W/k\rceil\right)\right)$. Then, the schedule needs to be constructed to determine the communication delays, which involves $O\left(mn\lceil W/k\rceil\right)$ scheduling events. A message arrival event inserts the message into deadline ordered message ready queue, which has the complexity of $O\left(lg\left(mn\lceil W/k\rceil\right)\right)$ using a binary heap. Now that messages in the ready queue are in deadline order, a scheduling decision only costs $O\left(1\right)$. Thus, the total cost of computing a message communication delay is :

$$O\left(mn\lceil W/k\rceil lg\left(mn\lceil W/k\rceil\right)\right) + O\left(mn\lceil W/k\rceil\right) \times O\left(lg\left(mn\lceil W/k\rceil\right)\right).$$

That cost becomes $O\left(mn\lceil W/k\rceil lg\left(mn\lceil W/k\rceil\right)\right)$.

Similar to calculating communication delays, there are up to $mn\lceil W/k\rceil$ subtask arrivals on a processor. Computing the arrival time of a subtask involves computing communication delay of its triggering message. Thus, constructing the subtask arrival list on a processor costs $O\left(m^2n^2\lceil W/k\rceil^2 lg\left(mn\lceil W/k\rceil\right)\right)$. Once the subtask arrival list is available, algorithm 4 is invoked to check whether the processor is overloaded. Again, the overload analysis process costs $O\left(mn\lceil W/k\rceil lg\left(mn\lceil W/k\rceil\right)\right)$, given a subtask arrival list. Therefore, the cost of *OverloadCheck* algorithm is dominated by the cost of constructing the subtask arrival list, which is $O\left(m^2n^2\lceil W/k\rceil^2 lg\left(mn\lceil W/k\rceil\right)\right)$.

Stage I allocation may fully replicate each worker subtask of a task during each period to satisfy its deadline. The replication process for a single subtask is implemented by calling *AssignProcessor* algorithm up to p times. Each *AssignProcessor* in turn may need to examine all p processors. Thus, the worse-case complexity of replication for a subtask is $O\left(m^2n^2p^2\lceil W/k\rceil^2 lg\left(mn\lceil W/k\rceil\right)\right)$. The total cost of stage I allocation becomes $O\left(m^3n^3p^2\lceil W/k\rceil^3 lg\left(mn\lceil W/k\rceil\right)\right)$.

Stage II allocation examines each subtask during each period and checks if its fault-tolerance ability is maximized. In the worst case, every processor fails independently. Thus, maximum fault-tolerance may require full replication. Allocating a replica to a processor q during stage II involves searching for a spare time slot. In the worst case, all subtasks can have replicas on q. Therefore, the searching operation has the worst-case complexity of $O\left(mn\lceil W/k\rceil\right)$. Then the total cost of stage II allocation becomes $O\left(m^2n^2p\lceil W/k\rceil^2\right)$.

Therefore, the worst-case complexity of the BEA algorithm is the total cost of stage I and stage II allocation, which is $O\left(m^3n^3p^2\lceil W/k\rceil^3 lg\left(mn\lceil W/k\rceil\right)\right)$.

6 Experimental Evaluation of the BEA Algorithm

To evaluate how well BEA performs, we compare its performance with the full replication strategy, where all subtasks of all tasks during each period are fully replicated. Since we consider real-time and fault-tolerance as two orthogonal dimensions of QoS requirements, the performance of the algorithms are compared in terms of these two dimensions. In general, the more replicas the better. However, full replication may be too expensive to implement or even impossible. For example, some subtasks may require special run-time support from the host machine, e.g. special library, etc., which may not be available on all host machines. Thus, we are interested in determining how well BEA performs compared with full replication and to which extent BEA can save system resources while maintaining comparable performance.

The experimental system consists of 15 processors connected by 100Mbps fast

Table 1
Example task set parameters

ID	Type	#Subtasks	Deadline	$maxRTB_i$	I_i
0	periodic	4	0.7 s	20	3
0	aperiodic	4	0.45 s	10	2
1	periodic	4	0.28 s	18	5
1	aperiodic	4	0.33 s	5	10
2	periodic	4	1.2 s	15	4
2	aperiodic	4	1.1 s	3	8

Ethernet switch. The processor set is partitioned into three groups (3, 9, and 3 processors within each group) with a mean time to failure (MTTF) 10 s, 100 s, and 1000 s, respectively. The exponential failure-time model was used as an example model for specifying the reliability functions.

Table 1 summarizes the parameters of an example task set used in the experiments. Other task sets led to the similar conclusions, and hence are omitted here. We considered adaptation functions with an increasing ramp periodic load and aperiodic load, denoted as "ramp/ramp" load. The aperiodic load is fixed at 80% of the corresponding periodic load.

6.1 Fault-Tolerance Ability

Recall that the tasks considered in this paper are executed in serial. Thus, the reliability of a task T_i during period c is the smallest reliability among all of its subtasks. That is,

$$Reliability\,(T_i, c) = min\{Reliability\left(st_j^i, c\right)|st_j^i \in ST\,(T_i)\}. \tag{1}$$

However, the reliability of a subtask is the reliability acquired by executing the replicas of the subtask in parallel:

$$Reliability\left(st_j^i, c\right) = 1 - \prod_r \left(1 - Reliability\left(Host\left(r\right), c\right)\right). \tag{2}$$

For comparing fault-tolerance ability, we define the aggregate fault-tolerance ability of the task set during the future adaptation time window as the weighted sum of the task reliability:

$$FT = \sum_{T_i, c} \left(Reliability\,(T_i, c) \times I_i\right). \tag{3}$$

Figure 4 shows the aggregate fault-tolerance ability of BEA, as percentages of the aggregate fault-tolerance ability by full replication. Recall that fault-tolerance ability achieved by an algorithm is defined in Equation 3. As we can see, the fault-tolerance ability achieved by BEA is almost the same as that of the full replication— over 98% in the experiments. This is because of the host group model, where more than one replica within a host group does not improve the fault-tolerance ability.

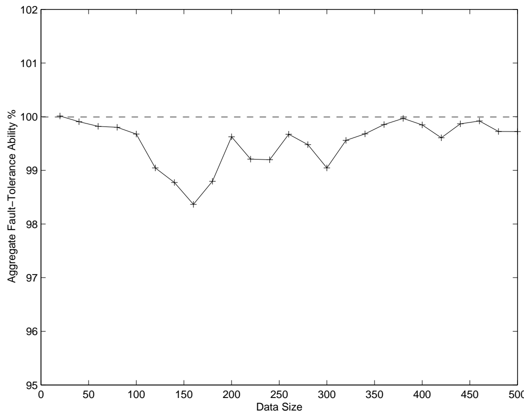

Fig. 4. Aggregate Fault-Tolerance Ability of BEA

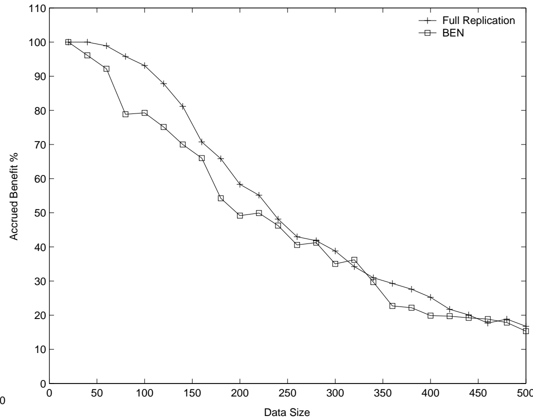

Fig. 5. Aggregate Real-Time Benefit

6.2 Real-Time Benefit

We also plot the percentages of aggregate real-time benefits accrued by BEA and full replication in Figure 5. Note that we assume step benefit functions for all tasks. Thus, completing an end-to-end task T_i before its deadline can accrue $maxRTB_i$ real-time benefit (the height of T_i's benefit function), while a deadline miss yields zero real-time benefit. For the sake of fair comparison, we use percentage of aggregate real-time benefit, which is the ratio of accrued real-time benefit to the maximal possible real-time benefit, i.e., sum of all $maxRTB_i$.

In general, full replication can accrue more real-time benefit than BEA. However, observe that the performance gap in Figure 5 is not so significant. Furthermore, the aggregate real-time benefits of BEA and full replication converge when the load increases. Interestingly, at some heavy load points, we observe that BEA is slightly better than full replication in terms of aggregate real-time benefit. We conjecture that full replication may produce too much interference on the processors, as more replicas will compete for processor time than BEA. On the contrary, BEA seeks to find the right number of replicas and their processor assignment, which is not necessarily the same as full replication. Furthermore, full replication may also increase the network traffic, due to the overhead associated with communication, e.g. message header etc.

Thus, we observe that BEA can accrue almost the same fault-tolerance ability and aggregate real-time benefit as full replication in the experiments. However, BEA may need significantly fewer replicas.

6.3 Number of Replicas

Figure 6 shows the average number of replicas by BEA. Ideally, every subtask should have three replicas residing in the three host groups, and hence the ideal average number of replicas is three. This ideal number is achieved by BEA at the beginning of the plot, i.e., during light load situation. When the workload increases, BEA determines three replicas may not be enough to satisfy the task deadlines. Thus, the algorithm replicates subtasks, which increases the average number of replicas.

During heavy load situations, however, BEA has to sacrifice some tasks to accrue more aggregate benefit. As we can see, the average number of replicas becomes less than three when the input data size is larger than 320. Compared with full replication(15 replicas in the experiments), BEA is found to be more cost effective, while maintaining comparable performance.

To reenforce the adaptation mechanism embedded in BEA, we show the number of replicas of a periodic task during each period in the experiments in Figure 7. As we expect, the number of replicas grows with the increase of period number. Recall that we used ramp/ramp load pattern in the experiments. Thus, workload in the later periods becomes heavier. It is possible that the algorithm decides to de-allocate this task to accrue more aggregate benefit during heavy load. Thus, we also observe that the number of replicas begins to decrease at the end of experimental time interval.

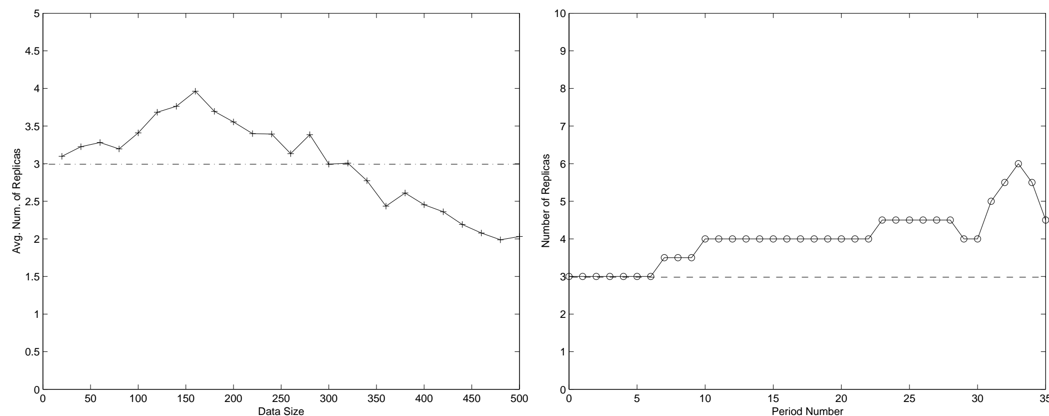

Fig. 6. Avg. Num. of Replicas by BEA Fig. 7. Num. Replicas of A Periodic Task

7 Conclusions

In this paper, we present a proactive, quality of service (QoS) allocation algorithm called BEA, for fault-tolerant asynchronous real-time distributed systems. BEA considers an application model where trans-node application timeliness are expressed using benefit functions, and anticipated workload during future time intervals are expressed using adaptation functions. Furthermore, BEA considers an adaptation model where subtasks of application tasks are replicated at run-time for tolerating failures as well as for sharing workload increases. Given such models, the objective of the algorithm is to maximize the aggregate real-time benefit and fault-tolerance ability during the time window of adaptation functions.

Since this resource allocation problem is NP-hard, BEA heuristically computes near-optimal resource allocations in polynomial-time. To determine how well BEA performs, we compare the performance of BEA with that of the full replication strategy. We show that BEA can achieve almost the same fault-tolerance ability as full replication, and most of the aggregate real-time benefit accrued by full replication. However, BEA needs significantly fewer replicas, and thus is cost effective.

Therefore, the major contribution of the paper is the BEA algorithm that seeks to maximize aggregate real-time benefit and fault-tolerance ability in asynchronous real-time distributed systems.

Acknowledgements

This paper is a revised and expanded version of our original paper with the same title, which appears in the *Proceedings of* 7th *IEEE/IEICE International Symposium on High Assurance Systems Engineering*, pp. 19-26.

References

Abdelzaher, T., Shin, K., June 1998. End-host architecture for qos-adaptive communication. In: Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium. pp. 121–130.

Bettati, R., December 1997. Proceedings of The IEEE Workshop on Middleware for Distributed Real-Time Systems and Services. IEEE Computer Society, the 18th IEEE Real-Time Systems Symposium.

Birman, K. P., December 1993. The process group approach to reliable distributed computing. Communications of the ACM 36 (12), 127–133.

Brandt, S., Nutt, G., Berk, T., Mankovich, J., December 1998. A dynamic quality of service middleware agent for mediating application resource usage. In: Proceedings of The IEEE Real-Time Systems Symposium. pp. 307–317.

Clark, R., Jensen, E. D., Kanevsky, A., Maurer, J., Wallace, P., Wheeler, T., Zhang, Y., Wells, D., Lawrence, T., Hurley, P., April 1999. An adaptive, distributed airborne tracking system. In: Proceedings of The Seventh IEEE International Workshop on Parallel and Distributed Real-Time Systems. Vol. 1586 of Lecture Notes in Computer Science. Springer-Verlag, pp. 353–362.

Clark, R. K., 1990. Scheduling dependent real-time activities. Ph.D. thesis, Carnegie Mellon University, cMU-CS-90-155.

DARPA, August 1997. Quorum. `http://www.darpa.mil/ipto/research/quorum/index.html`.

Hegazy, T., September 2001. Using application benefit for proactive resource allocation in asynchronous real-time distributed systems. Master's thesis, Virginia Tech.

Hegazy, T., Ravindran, B., August 2002. Using application benefit for proactive resource allocation in asynchronous real-time distributed system. IEEE Transactions on Computers 51 (8), 945–962.

HiPerD, 1997. High performance distributed computing. `http://www.nswc.navy.mil/hiperd/index.shtml`.

Jensen, E. D., October 1992. Asynchronous decentralized real-time computer systems. In: Halang, W. A., Stoyenko, A. D. (Eds.), Real-Time Computing. Proceedings of the NATO Advanced Study Institute. Springer Verlag.

Jensen, E. D., Ravindran, B., August 2002. Guest editor's introduction to special section on asynchronous real-time distributed systems. IEEE Transactions on Computers 51 (8), 881–882.

Kao, B., Garcia-Molina, H., December 1997. Deadline assignment in a distributed soft real-time system. IEEE Transactions on Parallel and Distributed Systems 8 (12), 1268–1274.

Koob, G., October 1996. Quorum. In: Proceedings of The Darpa ITO General PI Meeting. pp. A–59–A–87.

Lee, C., August 1999. On quality of service management. Ph.D. thesis, Carnegie Mellon University.

Li, P., Ravindran, B., Hegazy, T., November 2001. Implementation and eveluation of a best-effort scheduling algorithm in an embedded real-time system. In: IEEE International Symposium on Performance Analysis of Systems and Software. Tucson, AZ, pp. 22–29.

Mills, D. L., June 1995. Improved algorithms for synchronizing computer network clocks. IEEE/ACM Transactions on Networks , 245–254.

MK7.3a, October 1998. MK7.3a Release Notes. The Open Group Research Institute's Real-Time Group, Cambridge, Massachusetts.

Ravindran, B., January 2002. Engineering dynamic real-time distributed systems: Architecture, system description language, and middleware. IEEE Transactions on Software Engineering 28 (1), 30–57.

Rosu, D., Schwan, K., Yalamanchili, S., Jha, R., December 1997. On adaptive resource allocation for complex real-time applications. In: Proceedings of The 18th IEEE Real-Time Systems Symposium. pp. 320–329.

Shirazi, B., Welch, L. R., Ravindran, B., Cavanaugh, C., Huh, E., March 2000. Dynbench: A benchmark suite for dynamic real-time systems. Journal of Parallel and Distributed Computing Practices 3 (1), 89–107.

Welch, L. R., Ravindran, B., Shirazi, B., Bruggeman, C., December 1998. Specification and modeling of dynamic, distributed real-time systems. In: Proceedings of The IEEE Real-Time Systems Symposium. pp. 72–81.