



Available at

www.ElsevierComputerScience.com

POWERED BY SCIENCE @ DIRECT®

The Journal of Systems and Software xxx (2003) xxx–xxx

www.elsevier.com/locate/jss

Proactive QoS negotiation in asynchronous real-time distributed systems [☆]

Peng Li ^{*}, Binoy Ravindran ¹

Real-Time Systems Laboratory, Department of ECE, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA

Received 9 September 2002; received in revised form 10 December 2002; accepted 23 December 2002

Abstract

We present a fast, proactive, quality of service (QoS) negotiation algorithm called Best Effort Negotiation (or BEN), for asynchronous real-time distributed systems. BEN considers an application model where trans-node application timeliness and fault-tolerance requirements are expressed using benefit functions, and anticipated workload and system failure rates during future time intervals are expressed using adaptation functions and reliability functions, respectively. Furthermore, BEN 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 sum of aggregate real-time and fault-tolerance benefits during the time window of adaptation functions. Since determining the optimal solution is computationally intractable, BEN heuristically computes sub-optimal resource allocations in polynomial-time. To determine how well BEN performs, we describe another algorithm called HLC, that is inspired by the well-known Hill Climbing heuristic. We show that HLC is significantly slower than BEN. However, our experimental studies reveal that the performance of BEN, in general, is as good as that of HLC.

© 2003 Published by Elsevier Inc.

Keywords: 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. Thus,

they violate the deterministic foundations of hard real-time theory that ensures that “all timing constraints are always satisfied” under deterministic postulations of application workloads and execution environment characteristics.

To deal with such non-determinism’s, 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 applications’ 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.

[☆] This work was supported by the US Office of Naval Research under Grant N00014-99-1-0158 and N00014-00-1-0549.

^{*} Corresponding author. Tel.: +1-540-231-1418; fax: +1-540-231-3362.

E-mail addresses: peli2@vt.edu (P. Li), binoy@vt.edu (B. Ravindran).

¹ Tel.: +1-540-231-3777; fax: +1-540-231-3362.

Examples of such real-time QoS techniques are presented in Abdelzaher and Shin (1998), Brandt et al. (1998), Lee (1999), Ravindran (2002a) and 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 re-allocate 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 sub-optimal resource allocation decisions, many reactive resource allocation techniques such as Ravindran (2002a) and Rosu et al. (1997) sacrificed 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 QoS negotiation algorithm called Best Effort Negotiation (or BEN), for asynchronous real-time distributed systems. BEN is proactive in the sense that it enables *user-triggered*, *user-specified*, *application-specific*, and *situation-specific* resource allocation.

BEN considers an application model where trans-node application tasks have end-to-end timeliness and fault-tolerance requirements that are expressed using benefit functions. Furthermore, to facilitate proactive QoS negotiation, BEN uses *adaptation functions*—a concept that we had developed in Hegazy and Ravindran (2002b)—for describing the anticipated application workload during future time intervals. Similar to the workload adaptation functions, the algorithm considers reliability functions that probabilistically describe the failure rates of end-host groups, during future time intervals.

BEN 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. Furthermore, the algorithm targets shared-media broadcast networks such as Gigabit Ethernets that run the Carrier Sense Multi Access/Deadline Driven Collision Resolution (CSMA/DDCR) protocol (Hermant and Lann, 1998), and interconnect end-hosts that 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 BEN is to maximize the aggregate real-time and fault-tolerance benefits of the application during future time intervals described by the adaptation and reliability functions. This problem of

maximizing aggregate benefits can be shown to be \mathcal{NP} -hard.² Thus, BEN is a heuristic algorithm for this problem that produces sub-optimal resource allocations in polynomial-time.

To study how well BEN performs, we consider another algorithm called HLC (for Hill Climbing), that is inspired by the well-known Hill Climbing heuristic strategy. HLC determines resource allocations by searching a large solution-space, due to the nature of the underlying Hill Climbing technique.

Given n application tasks, we find that the worst-case computational complexity of BEN is in the third-order of n , while that of HLC is in the fourth-order of n . Furthermore, our application-driven experimental studies reveal that the performance of BEN, in general, is close to that of HLC. Thus, BEN is as good as HLC, but for a significantly less computational cost.

Therefore, the major contribution of the paper is the polynomial-time BEN algorithm that seeks to maximize aggregate real-time *and* fault-tolerance benefits 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 QoS negotiation and resource allocation problem that we are addressing in this paper. Sections 4 and 5 present BEN and HLC, respectively. We present the computational complexity of BEN and HLC in Section 6. In Section 7, we discuss the experimental evaluation of the algorithms. Finally, the paper concludes with a summary of the work and its contributions in Section 8.

2. Application, adaptation, system, and failure models

2.1. The application model

We denote the set of tasks in the application by the set $T = \{T_1, T_2, T_3, \dots\}$, 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 . 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 $T_i = [st_1^i, m_1^i, st_2^i, m_2^i, \dots, st_n^i, m_n^i]$ to represent a task T_i that consists of n subtasks and n messages to be executed and transmitted in series, such that subtask

² For brevity, we do not prove this here. However, the proof is similar to the one shown in Hegazy (2001) for a similar resource allocation problem.

$st_j^i (1 < j \leq n)$ cannot execute before message m_{j-1}^i arrives. For convenience, we denote the set of subtasks of a task T_i as $ST(T_i) = \{st_1^i, st_2^i, st_3^i, \dots, st_n^i\}$ and the set of inter-subtask messages of a task T_i as $MS(T_i) = \{m_1^i, m_2^i, m_3^i, m_4^i, \dots, m_n^i\}$.

2.1.1. Timing requirements

We use Jensen's benefit functions (Jensen, 1992) for expressing application timeliness requirements, and assume "step" benefit functions for all tasks such as the one shown in Fig. 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 $\max RTB_i$.

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

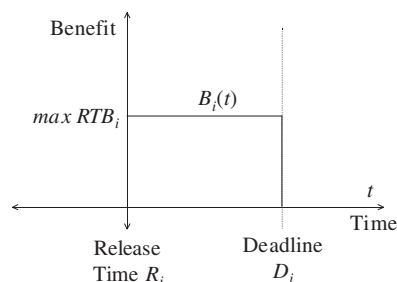


Fig. 1. A task benefit function.

2.2. The adaptation model

2.2.1. Adaptation mechanism

We assume that subtasks of tasks can be dynamically and transparently *replicated* for tolerating end-host failures. Furthermore, we assume that the subtasks can be replicated for sharing workload increases. That is, once a subtask is replicated, the replicas can be executed on different end-host processors, and the workload of the original subtask can be distributed among the replicas. This will enable concurrency to be exploited in processing the workload and reduce (end-to-end) task latency. Thus, replication is used as a mechanism for tolerating end-host failures, as well as a mechanism for adapting the application to workload fluctuations.

Of course, when subtasks are replicated and the workload is shared among the replicas, this may increase communication latencies on shared media broadcast networks—our target platforms—due to the increased contention that the messages may now experience (since there are additional messages that now flow into, and out of the replicas). Thus, an important objective of the negotiation algorithms is to replicate subtasks (1) only to the extent that any decreases in subtask execution latencies are not offset by increases in communication latencies, and (2) sufficiently to the extent that the aggregate application benefit can be maximized as much as possible under fluctuating workloads.

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.

We assume that the subtasks process data objects (or sensor reports) 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 (Jensen, 1992). The subtask replicas are thus assumed to be *temporally consistent* without applying every change in value, due to the continuity of the physical phenomena.

2.2.2. Adaptation functions

We use adaptation functions, a concept that we originally developed in Hegazy and Ravindran (2002b), 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. We denote the anticipated workload of a task T_i during a period p as $Adapt(T_i, p)$. 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 Fig. 2.

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 QoS negotiation and resource allocation.

Thus, adaptation functions facilitate a “human-in-the-loop” approach, where adaptation of the application can be performed according to the 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 observe 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. With proactive resource allocation, it will be possible to compute sub-optimal resource allocation decisions due to the longer time interval available for resource allocation as they are performed before “event arrival”. (Of course, we desire very fast algorithms.) However, reactive resource allocation is still needed, especially when the user-specified future workload and failure scenarios deviate from the actual i.e., to adapt the system when the user’s guesses wrong. Furthermore, in many situations, workload changes and failure occurrences can be so dynamic and arbitrary that they prevents human intervention in the control loop. Therefore, we believe that adaptive resource allocation that is proactively *and* reactively performed will enhance system utility.

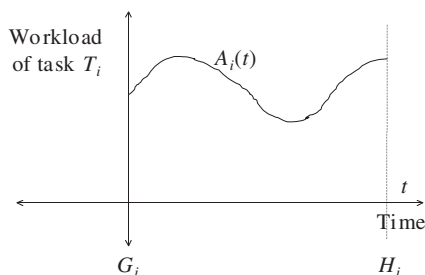


Fig. 2. A task adaptation function.

2.3. The system model

We consider a Local Area Network segment where end-hosts are interconnected using a shared media broadcast network such as Gigabit Ethernet that runs the CSMA/DDCR protocol (Hermant and Lann, 1998). Thus, physical frames of application messages are scheduled (by CSMA/DDCR) using non-preemptive Earliest Deadline First (EDF) algorithm (Stankovic et al., 1998).

We denote the set of end-host processors by the set $PR = \{p_1, p_2, p_3, \dots, p_m\}$. 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. 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 available in operating systems including the MK7.3 kernel (MK7.3a, 1998) and our own Linux implementation (Li et al., 2001).

2.4. The failure model and fault-tolerance benefit

We assume a fail-stop model for the end-hosts, i.e., a host simply halts when it fails. Similar to the workload adaptation functions, we consider reliability functions for end-host groups. For specifying reliability functions, we consider a *logical* partition $\{H_i\}_{i \in \{1, \dots, g\}}$ of the host set H into g groups such that $\forall i, H_i \neq \emptyset$, $\bigcup_{i=1}^g H_i = H$, and $\bigcap_{i=1}^g H_i = \emptyset$. 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. Furthermore, when subtasks are replicated, each replica of a subtask processes a unique portion of the workload of the subtask. Thus, 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 thus temporally consistent without applying every change in value, due to the continuity of physical phenomena.

Following Jalote (1994), we define the reliability function $R(H_i, t_\alpha, t_\beta)$ to denote the probability that $\forall j, h_j \in H_i$ will *not* fail during a time interval $[t_\alpha, t_\beta]$. Reliability functions are user-specified, can have arbitrary shapes, and can be dynamically modified as and when the user's perception regarding anticipated host-group failures change. Note that we do not make any assumptions on the failure rate or any particular failure-time model for specifying the reliability of host groups. The function $R(H_i, t_\alpha, t_\beta)$ must however, satisfy the basic properties of a reliability function i.e., $R(H_i, t, t) = 1$, $\lim_{t_\beta \rightarrow \infty} R(H_i, t_\alpha, t_\beta) = 0$, and $R(H_i, t_1, t_2) \geq R(H_i, t_1, t_3)$, iff $t_2 \leq t_3$.

We define the fault tolerance benefit that is gained by a task T_i , when a single replica is allocated for each subtask of the task as the Fault-Tolerance benefit Factor (FTF_i) of the task. The fault-tolerance benefit factor is user-defined for all tasks. In general, the fault-tolerance benefit of a task increases with the number of replicas that are allocated to subtasks of the task. However, the fault-tolerance benefit of a task is also affected by the end-host assignment of the subtask replicas of the task. Therefore, we define the Fault-Tolerance Benefit of subtask st_j^i during the time interval $[t_1, t_2]$ as $FTB_j^i = FTF_i \times \sum_{k=1}^{r_j} R(h_k, t_1, t_2)$, where r_j is the number of replicas of subtask st_j^i and h_k is the end-host executing the k th replica of the subtask during the interval $[t_1, t_2]$.

Note that the subtasks of a task can have different number of replicas and end-host assignments and hence, can have different fault-tolerance benefits. Thus, we define the task-level fault-tolerance benefit FTB_i as the smallest value among all its subtask fault-tolerance benefits, since the subtasks are assumed to “serially” execute. Therefore, the fault-tolerance benefit of a task T_i is defined as $FTB_i = \min\{FTB_j^i, 1 \leq j \leq |ST(T_i)|\}$.

Furthermore, it is important to notice that the BEN algorithm we propose here is not tightly coupled to any specific fault-tolerance benefit definition. Given a resource allocation, i.e., number of replicas for each subtask and their end-hosts assignment, the BEN algorithm simply uses the fault-tolerance benefit definition and the task benefit functions to compute the system-level aggregate benefit. We show how BEN computes the aggregate benefit in Section 4.3. Thus, our definition of the fault-tolerance benefit only serves as an example definition.

Besides task fault-tolerance benefit factors, we also consider a user-specified, required, lower bound on the number of replicas for subtasks of tasks. For a task T_i , this minimum required number of replicas is denoted as $\min r_i$.

As an example output of the resource allocation algorithm (see Fig. 3), we consider six end-hosts, $PR = \{p_1, p_2, \dots, p_6\}$, connected using a CSMA/DDCR real-time Ethernet network, and two tasks, T_i and T_j . Task T_i has four subtasks, while task T_j has two sub-

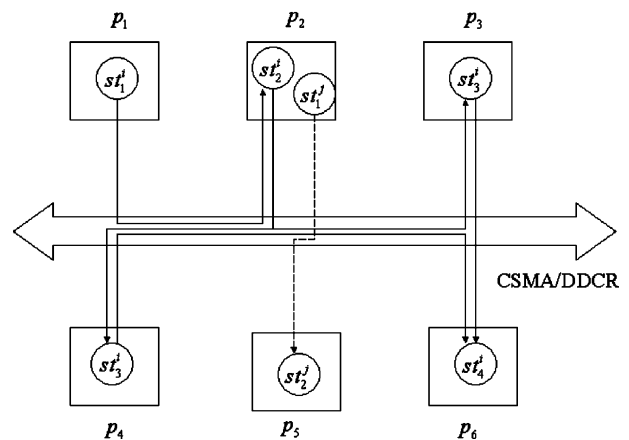


Fig. 3. A resource allocation example.

tasks. As shown in the figure, the resource allocation algorithm allocates one replica of st_1^i on host p_1 , one replica of st_2^i on host p_2 , two replicas of st_3^i on hosts p_3 and p_4 , and one replica of st_4^i on host p_6 . Similarly, the two subtasks of task T_j are not replicated and are assigned to hosts p_2 and p_5 , respectively. Thus, during any given time interval $[t_1, t_2]$, the subtask-level fault tolerance benefit of st_3^i is calculated as $FTB_3^i = FTF_i \times (R(p_3, t_1, t_2) + R(p_4, t_1, t_2))$. On the other hand, we calculate the fault-tolerance benefit of other subtasks of T_i as $FTB_1^i = FTF_i \times R(p_1, t_1, t_2)$, $FTB_2^i = FTF_i \times R(p_2, t_1, t_2)$, and $FTB_4^i = FTF_i \times R(p_6, t_1, t_2)$. Therefore, the task-level fault-tolerance benefit of T_i is computed as $FTB_i = \min\{FTB_k^i, k = 1, \dots, 4\}$. The same calculation procedure applies to task T_j as well.

3. QoS negotiation and resource allocation objective

Given the application, adaptation, failure, and system models described in Section 2, our objective is to maximize the aggregate real-time and fault-tolerance benefits during the time window of task adaptation functions and host-group reliability functions. We define the aggregate real-time and fault-tolerance benefits of the application as the sum of the real-time and fault-tolerance benefits accrued by all tasks of the application.

Thus, our QoS negotiation and resource allocation objective can be informally stated as follows

Consider the application, adaptation, failure, and system models described in Section 2. Given these models where each task can have an *arbitrary workload*, what allocation—number of replicas for each subtask and their end-hosts—will maximize the aggregate real-time and fault-tolerance benefits during the time window of the task adaptation functions and host-group reliability functions when end-hosts may *fail arbitrarily*?

This problem can be shown to be \mathcal{NP} -hard by mapping it to the multiple 0–1 Knapsack problem, as stated in Section 1. Thus, the BEN and HLC algorithms that we present in this paper are heuristic algorithms that solve this problem in polynomial-time, but not necessarily produce optimal allocations.

4. The BEN algorithm: heuristics and rationale

The objective of BEN is to maximize the sum of the aggregate real-time and fault-tolerance benefits while the minimum replica needs of all tasks (i.e., $\min_{\mathcal{R}_i} \forall T_i$), are satisfied. Therefore, the desired properties of BEN include:

- (1) *Allocate the minimum required number of replicas for each task before allocating additional replicas.* This initial allocation guarantees that the minimum replica requirement of each task is satisfied.
- (2) *Allocate resources in decreasing order of $\max RTB_i + FTF_i$.* By doing so, we increase the possibility of maximizing the sum of the aggregate real-time and fault-tolerance benefits. This is because, the task selected next for resource allocation is always the one with the highest sum of real-time and fault-tolerance benefits among the unallocated tasks.
- (3) *Proceed to the next task adaptation period for resource allocation, if replicating the task during the current period cannot further increase the sum of the aggregate real-time and fault-tolerance benefits.* By doing so, we save system resources, which can be allocated to tasks with lower values of $\max RTB_i + FTF_i$. This will increase the possibility of obtaining contributions of non-zero sum of real-time and fault-tolerance benefits from such tasks toward the sum of aggregate real-time and fault-tolerance benefits;
- (4) *Decompose task-level resource allocation problem into subtask-level resource allocation problems.* The rationale behind this heuristic is that solving the task-level resource allocation problem of determining the replica needs of subtasks of a task and their end-hosts that will satisfy the task deadline, 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 sub-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 that will satisfy the task deadline by determining the replica needs of subtasks of the task that will satisfy the subtask deadlines.

Thus, BEN allocates resources using the heuristics presented here. To assign deadlines to subtasks and messages of a task from the end-to-end task deadline, BEN uses the equal flexibility (EQF) algorithm presented in Kao and Garcia-Molina (1997). EQF assigns deadlines to subtasks and messages that are proportional to their execution times and communication delays, respectively. BEN therefore, uses an anticipated initial workload—workload that is anticipated to initially exist—to estimate the subtask execution times and message communication delays. The execution times and communication delays are then used to assign subtask and message deadlines, according to EQF.

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, BEN defines the real-time benefit of a subtask as simply the real-time benefit $\max RTB_i$ of its parent task.

Algorithm 1. [High level description of BEN]

- 1: **Input:** a set of tasks $T = \{T_1, T_2, \dots, T_n\}$ in descending order of $\max RTB_i + FTF_i$, and an adaptation window W ;
- 2: **for** each task $T_i \in T$ **do**
- 3: Allocate $\min_{\mathcal{R}_i}$ replicas of T_i in round-robin fashion;
- 4: **for** each task $T_i \in T$ **do**
- 5: **for** each period $j = 1$ to $\lceil \frac{W}{\text{period}(T_i)} \rceil$ **do**
- 6: $QoS_Allocation(T_i, j, Adapt(T_i, j))$;

Algorithm 1 shows the high-level pseudo-code of the BEN algorithm. BEN first determines the resource allocation that will satisfy the minimum replica requirements of tasks in decreasing order of their $\max RTB_i + FTF_i$ values. To efficiently determine the next task for resource allocation, the algorithm constructs a heap for the task set, which has $\max RTB_i + FTF_i$ as key values of the heap nodes. BEN then allocates the minimum number of replicas for subtasks of a task in the order of their precedence relationships i.e., from subtask st_1^i to subtask st_m^i . The algorithm selects the end-hosts of the replicas in decreasing order of end-host reliabilities (i.e., from the end-host with the longest mean-time-to-failure to the end-host with the shortest mean-time-to-failure) in a round-robin manner.

Once the initial allocation is completed, BEN further allocates replicas to tasks to maximize the sum of aggregate benefits. Note that this allocation problem is different from the initial allocation problem, as here,

BEN needs to determine the number of replicas of each subtask of each task and their end-hosts that will maximize the sum of aggregate benefits. In the initial allocation problem, the number of replicas of each subtask is *known*; the algorithm only needs to determine the end-hosts of the replicas.

To maximize the sum of aggregate benefits, BEN again allocates replicas to tasks in decreasing order of $\max RTB_i + FTF_i$. Since the anticipated workload may be different for different task periods in the time window specified by the adaptation functions, BEN allocates replicas for each period in the time window, starting from the most recent period and proceeding to the least recent. The allocation procedure for each task during each period is shown in Algorithm 2.

Algorithm 2. [*QoS_Allocation()*]

```

1: Input:  $T_i$  and its adaption function at period  $j$ ,
    $Adapt(T_i, j)$ 
2: while (true) do
3:    $SB = ComputeBenefit(T_i, j)$ ;
4:   for each subtask  $k = 1$  to  $|ST(T_i)|$  do
5:     Tentatively allocate one or more replicas to  $st_k^i$ ;
     by calling  $Assignprocessor(st_k^i, Adapt(T_i, j), j)$ ;
6:      $New\_SB = ComputeBenefit(T_i, j)$ ;
7:     if  $New\_SB > SB$  then
8:       Accept the tentative allocation;
9:     else
10:      De-allocate the tentative allocation;
11:    break;

```

For each task T_i during each period, the *QoS_Allocation* algorithm allocates replicas for subtasks of the task starting from the first subtask and proceeding to the last subtask. For each subtask, the algorithm tentatively allocates a single replica and selects an end-host processor for the replica by analyzing processor overloads. The algorithm then checks whether the tentative allocation increases the sum of aggregate benefits. If it does, the allocation is considered as a “good” allocation and the algorithm further replicates the subtask. The process is repeated (as long as the sum of aggregate benefits increases) until the maximum possible number of replicas for a subtask is reached. This maximum limit is the number of end-host processors in the system, as that limits the maximum concurrency that

can be exploited as well as failures that can be tolerated. Note that in case of a subtask deadline miss, no real-time or fault-tolerance benefit can be accrued. Thus, the tentative allocation may need to allocate more than one replica to a subtask, such that its individual deadline can be satisfied before checking the sum of benefits.

If the tentative allocation at any step during the resource allocation process does not increase the sum of aggregate benefits, the algorithm de-allocates the tentative allocation and proceeds to the next task adaptation period.

We show an example of the *QoS_Allocation()* algorithm for periodic task T_i and its corresponding aperiodic task during its j th allocation period in Table 1. We assume that periodic task T_i has two subtasks and aperiodic task T_i has three subtasks. Furthermore, we assume that every subtask of T_i (both periodic and aperiodic) has been assigned an end-host during the initial allocation, as shown in row 4 of Table 1. The *QoS_Allocation()* algorithm first tentatively allocates one more replica for every subtask of the periodic and aperiodic tasks T_i , which increases the aggregate fault-tolerance and real-time benefit to 24 (Table 1, row 5). Thus, this tentative allocation is accepted. The *QoS_Allocation()* algorithm then seeks to allocate another replica to every subtask of T_i . Although only one under-loaded end-host is found for subtask st_1^i of aperiodic task T_i , the aggregate benefit is still increased by 16 (row 6). Therefore, the allocation shown in row 6 is accepted as well. Finally, the algorithm finds an under-loaded end-host for aperiodic subtask st_2^i in row 7, which, however, decreases the aggregate benefit to 36. Thus, the algorithm de-allocates the last allocation and proceeds to the $(j + 1)$ th allocation period of task T_i .

We now discuss how BEN (1) analyzes processor overloads to select end-hosts for subtask replicas, and (2) how the algorithm computes the sum of aggregate benefits of a tentative resource allocation to determine whether the allocation is “good”. These steps are discussed in the subsections that follow.

4.1. Analyzing processor overloads

BEN analyzes end-host processor overloads to select an end-host processor for a subtask replica. The rationale

Table 1
QoS_Allocation() example

Task T_i at allocation period j					Aggre. Benefit	Accept
Periodic		Aperiodic				
st_1^i	st_2^i	st_1^i	st_2^i	st_3^i	–	–
p_1	p_2	p_1	p_2	p_3	20	–
p_1, p_4	p_2, p_5	p_1, p_4	p_2, p_5	p_3, p_6	24	Yes
p_1, p_4	p_2, p_5	p_1, p_4, p_6	p_2, p_5	p_3, p_6	40	Yes
p_1, p_4	p_2, p_5	p_1, p_4, p_6	p_2, p_5, p_3	p_3, p_6	36	No

behind overload analysis is that if an end-host processor is under-loaded after assigning a subtask with a given workload to the processor, 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, where EDF guarantees all deadlines. Thus, if an end-host processor is under-loaded, we can conclude 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* procedure that selects a processor for a subtask replica is shown in Algorithm 3.

Algorithm 3. [*AssignProcessor*()]

- 1: **Input:** subtask st_j^i and its workload l at period c ;
- 2: **while** underloaded processor for st_j^i has not been found **do**
- 3: Extract the next most reliable processor q from the heap;
- 4: **if** $OverloadCheck(st_j^i, c, q, l/(NumReplicas(st_j^i) + 1)) \neq \text{true}$ **then**
- 5: **return** q ;
- 6: **return false**;

To test for a processor overload, BEN examines deadlines of all subtasks on the processor in their increasing order. For each subtask deadline, BEN compares the subtask deadline against the cumulative sum of the remaining execution times of all subtasks with lesser deadlines (than the subtask deadline). If the cumulative sum is less than the subtask deadline for *all* subtask deadlines, then it indicates that the total processor-time demand of the subtasks is less than the available processor-time and hence, the processor is under-loaded. On the other hand, if the cumulative sum is large than any *one* subtask deadline, it indicates that the demand exceeds the available processor-time and hence, the processor is overloaded.

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. During an under-load situation when all subtasks can meet their deadlines, the response time of a subtask under EDF is simply the sum of the execution times of all subtasks with deadlines less than or equal to that of the subtask. 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 subtasks on the processor must be known. For determining subtask arrival times, BEN assumes that a subtask completes its execution when all of its replicas complete their execution. Therefore, the response time of a subtask is the longest response time among all its replicas. Furthermore, 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.

Thus, the arrival time of a subtask can be determined as the sum of the response times of subtasks, the 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 $ArrivalTime(st_j^i) = ArrivalTime(T_i) + \sum_{k=1}^{j-1} [ResponseTime(st_k^i) + Delay(m_k^i)]$, where $ArrivalTime()$ denotes the arrival time of a subtask or a task, $ResponseTime()$ denotes the response time of a subtask, and $Delay()$ denotes the communication delay of a message.

As discussed previously, the response time of a subtask on a processor can be determined as part of the overload analysis. We discuss how BEN determines communication delays of messages in Section 4.2.

The arrival time of each subtask on an end-host processor can thus be determined and an arrival list can be constructed. BEN constructs the arrival list as a binary heap, where subtask deadlines are used as keys and arrival times are used as values for the heap nodes. This enables the algorithm to efficiently determine the earliest deadline subtask arrival time at any given time by performing an “Extract-Min” operation on the heap. BEN thus uses the arrival list heap to analyze processor overload situations.

Observe that BEN allocates resources for tasks in the decreasing order of their $maxRTB_i + FTF_i$ values. This order may be different from the decreasing order of the real-time benefit $maxRTB_i$ of the tasks, which are inherited by the subtasks. This can introduce errors in the overload analysis, because the scheduling algorithm is not aware of the fault-tolerance benefits. For example, a task having low $maxRTB_i + FTF_i$ can have high real-time benefit $maxRTB_i$, although the fault-tolerance benefit FTF_i is low. Subtasks of such tasks (with high $maxRTB_i$) can interfere with the execution of subtasks that belong to tasks with higher $maxRTB_i + FTF_i$ values, but lower $maxRTB_i$.

To reduce such errors, BEN always selects that under-loaded processor for a subtask replica from all under-loaded processors, that currently has the minimum number of subtask replicas with a lower real-time benefit $maxRTB_i$ than that of the subtask. By doing so, the algorithm seeks to minimize the possibility of introducing errors into the overload analysis.

Algorithm 4. [*OverloadCheck()*]

- 1: **Input:** subtask st_j^i , and its workload share w at period c on processor q ;
- 2: Compute the arrival subtask list ST_list on processor q ;
- 3: $Sum = 0$;
- 4: **for** each subtask $st_r^s \in ST_list$ in increasing order of arrival times **do**
- 5: $Sum = Sum + RemainingTime(st_r^s)$;
- 6: **if** $Sum > Deadline(st_r^s)$ **then**
- 7: **return true**;
- 8: **if** $st_r^s == st_j^i$ **then**
- 9: $ResponseTime = Sum$;
- 10: **return ResponseTime**;

Pseudo-code of the *OverloadCheck* procedure is shown in Algorithm 4. Note that *OverloadCheck* is called by *AssignProcessor* to check whether executing a subtask with a given workload on a processor will cause an overload on the processor.

4.2. Determining message communication delays

We consider the problem of computing the upper bound on the communication delay of messages that are sent out by *all* replicas of subtask st_s^i during task period p . The set of these messages is denoted as M . Following Hermant and Lann (1998), to establish the upper bound on the communication delay of M , we only need to consider the time interval between the arrival time of M and the absolute deadline of M . We indicate this time interval as $I(M)$. Thus, the communication delay incurred by a message $msg \in M$ under CSMA/DDCR protocol consists of three parts: (1) the *collision resolution time*, which is the amount of time spent by CSMA/DDCR to search the Time Tree and the Static Tree for resolving message collisions, (2) the actual *physical transmission time* of messages that may be transmitted before msg after the collision is resolved, and (3) the *physical transmission time* of msg itself.

The message delay computation procedure involves the following three parameters (Hermant and Lann, 1998):

- $r(M)$, the upper bound on the number of messages that will be transmitted before $msg \in M$ from the same processor, where msg is sent out. Since there could be several replicas of st_s^i executed on different

processors, we denote this upper bound number on processor h as $r_h(M)$, where $h \in H(st_s^i)$ is a processor executing one replica of st_s^i ;

- $u(M)$, the upper bound on the total number of messages that will be transmitted by all hosts during time interval $I(M)$;
- $uTime(M)$, the total physical transmission time of $u(M)$ messages transmitted at throughput ψ . Notice that M is a subset of the message set $u(M)$. Thus, $uTime(M)$ includes the time in part (2) and part (3) of the above communication delay.

In the original CSMA/DDCR feasibility condition, $r_h(M)$ and $u(M)$ are computed from a priori known upper bounds on arrival densities of messages. Since the upper bounds on message arrival densities are unknown in asynchronous systems (due to their non-deterministic workload conditions), BEN directly computes the above parameters so that the feasibility condition of Hermant and Lann (1998) can be used, and thus message communication delays. Observe that in the worst case, messages from each replica of each subtask during each period could potentially interfere the transmission of messages in M . Therefore, the message communication delay calculation iterates each replica of each subtask during each period and checks if their outgoing messages are either part of $u(M)$ or they directly contribute to $r_h(M)$. This check procedure is accomplished by message deadline and arrival time comparisons, which are variants of the original CSMA/DDCR feasibility conditions.

Following Hermant and Lann (1998), a message m_k^t arriving at processor h belongs to $r_h(M)$ if and only if $Deadline(m_k^t) \geq ArrivalTime(M)$ and $Deadline(m_k^t) \leq Deadline(M)$. Similarly, m_k^t belongs to $u(M)$ if and only if $Deadline(m_k^t) \geq ArrivalTime(M)$ and $Deadline(m_k^t) \leq Deadline(M) - TransTime(M)$, where $TransTime(M)$ denotes the transmission time of the message M .

Once all these parameters are computed, the tree search time can also be determined. Then the upper bound on the message communication delay can be established as the sum of $uTime(M)$ and the tree search time. For brevity, we omit the details in this paper. The detailed computation procedure can be found in Ravindran (2001).

When the communication delay of a message is determined, it is possible that the delay is larger than the message deadline, as the timeliness feasibility of the subtask messages were a priori guaranteed by CSMA/DDCR using the feasibility condition of the protocol. When this occurs, the computation procedure returns “OVERLOAD” to indicate the timing failure, which further results in a “true” return value in *OverloadCheck* procedure described in Algorithm 4. Thus, the aggregate benefit will be negatively affected and subtasks

may be de-allocated to resolve the network overload situation.

4.3. Computing the aggregate benefit

Algorithm 5 (*ComputeBenefit()*).

- 1: **Input:** task T_i and its allocation period p ;
- 2: $RTB = 0$; $FTB = 0$;
- 3: **for** each task T_i from T_i to all lower ($maxTRB + FTF$) tasks **do**
- 4: $lowBound = \lfloor p \times period(T_i) / period(T_i) \rfloor$;
- 5: $highBound = \lfloor (p + 1) \times period(T_i) / period(T_i) \rfloor$;
- 6: **for** each period $c = 1$ to $AdaptWindow / period(T_i)$ **do**
- 7: **for** each subtask $st_k^i \in ST(T_i)$ **do**
- 8: compute the response time of st_k^i ;
- 9: compute the fault-tolerance benefit FTB_k^i ;
- 10: **if** $c \in [lowBound, highBound]$ and all subtasks of T_i meet their deadlines **then**
- 11: $RTB = RTB + Adapt(T_i, c) \times maxRTB_i$;
- 12: $FTB = FTB + \min\{FTB_k^i, k = 1 \text{ to } |ST(T_i)|\}$;
- 13: **return** $RTB + FTB$;

Observe that BEN only needs to compare the sum of the aggregate real-time and fault-tolerance benefits before and after resource allocation during a task adaptation period. Thus, BEN uses a procedure called *ComputeBenefit* that accepts a task T_i and a period number p , and returns the sum of the aggregate real-time and fault-tolerance benefits of all lower $maxTRB + FTF$ tasks (including T_i) during that period p of task T_i . This is because higher benefit tasks are assumed not to be affected by T_i . Further, the sum of aggregate benefits during other time intervals remain the same as before and after the resource allocation. Furthermore, since tasks may have different periods, the time interval within period p of task T_i is bounded by $lowBound$ and $highBound$ in the algorithm.

In the meanwhile, because we are considering DASA, DASA will abort a subtask when it finds that the subtask misses its deadline. Under such a situation, no real-time or fault-tolerance benefit can be accrued by the subtask of the task, since part of the task will not be executed at all. Hence, only if a task can meet its deadline, does *ComputeBenefit* accumulate the sum of the real-time and fault-tolerance benefits of that task to the aggregate sum.

5. Hill Climbing as an alternative to BEN

To study the effectiveness of BEN, we consider the classical Hill Climbing heuristic search technique as an alternative strategy. For convenience, here we refer to the Hill Climbing strategy as HLC (for Hill Climbing).

Similarly to BEN, HLC first allocates replicas to subtasks to satisfy the minimum replica requirement. The algorithm then iteratively allocates replicas to subtasks to maximize the sum of aggregate benefits. Thus, the high level algorithm of HLC is the same as that of BEN and hence omitted here. However, during each iteration, HLC tries to allocate replicas to all tasks and selects the task, which will yield the maximum increase of the aggregate benefit if it is allocated Δ more replicas. We regard $\Delta \geq 1$ as a tunable parameter of the algorithm. This is different from BEN in that BEN only allocates replicas to the “current” task. Thus, HLC has a larger search space. Note that the algorithm may select the same task for resource allocation in successive steps. We show the HLC allocation procedure for each task in Algorithm 6, which should correspond to the *QoS_Allocation()* algorithm in the BEN algorithm (Algorithm 2).

6. Computational complexity of BEN and HLC

Algorithm 6. [*HLC_Allocation()*]

- 1: **Input:** T_i and its adaption function at period j , $Adapt(T_i, j)$
- 2: **while (true) do**
- 3: **for** each task $T_i \in T$ in decreasing order of ($maxTRB + FTF$) **do**
- 4: $lowBound = \lfloor j \times period(T_i) / period(T_i) \rfloor$;
- 5: $highBound = \lfloor (j + 1) \times period(T_i) / period(T_i) \rfloor$;
- 6: $SB = \sum_{c=lowBound}^{highBound} ComputeBenefit(T_i, c)$;
- 7: **for** each period $c \in [lowBound, highBound]$ **do**
- 8: tentatively allocate Δ more replicas for each subtask of T_i ;
- 9: $NewSB = \sum_{c=lowBound}^{highBound} ComputeBenefit(T_i, c)$;
- 10: $Increase(T_i) = NewSB - SB$;
- 11: de-allocate the tentative replicas;
- 12: select the task T_m with the maximum increase of aggregate benefit;
- 13: **if** $Increase(T_m) \geq Threshold$ **then**
- 14: accept the tentative allocation for T_m ;
- 15: **else**
- 16: **break**;

Given 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 , in Ravindran (2001), we show that the worst-case computational complexity of the BEN algorithm is

$$O(nlg(n) + mnp^2 \lceil W/k \rceil lg(p) + m^2 n^3 p \lceil W/k \rceil^3 lg(mn \lceil W/k \rceil)).$$

Furthermore, we show that of the HLC algorithm is

$$O(m^2 n^4 p^2 (n + p) \lceil W/k \rceil^3 lg(mn \lceil W/k \rceil) + mn^3 p^3 \lceil W/k \rceil^2 lg(p)).$$

For brevity, we do not show the analysis here. Details of the analysis can be found in Ravindran (2001).

Thus, the complexity of HLC is a fourth-order polynomial, while that of BEN is a third-order polynomial. Hence, the speed up of BEN over that of HLC is quite significant. This is not surprising as HLC searches a significantly large solution space for determining resource allocation decisions when compared to that of BEN.

7. Experimental evaluation of BEN and HLC algorithms

We conducted application-driven simulation studies to evaluate the performance of the algorithms. The simulation system is developed using the OMNeT++ toolkit (Varga, 2001). Each OMNeT++ model (a simulated system) consists of hierarchically nested modules, which may interact with each other by sending and receiving messages. Modules at the lowest level of the module hierarchy (called “simple modules”) contain the user-provided algorithms, and model indivisible components in the simulated system. Furthermore, the simple modules appear to run in parallel during simulation executions, because they are implemented by OMNeT++ as “coroutines” (similar to “threads”). Thus, OMNeT++ provides an ideal platform to simulate distributed systems.

We considered adaptation functions with two workload patterns: (1) a constant periodic load with an increasing ramp aperiodic load, denoted as “const/ramp” load, and (2) an increasing ramp periodic load with an increasing ramp aperiodic load, denoted as “ramp/ramp” load. A sample 5-task set is initiated in the system, whose parameters are derived from *DynBench* (Welch et al., 1998) and shown in Table 2. We also considered a partition of a set of 15 end-host machines into three groups (3, 9 and 3 hosts for each group) using three different reliability functions. The exponential failure-time model was used as an example model for specifying the reliability functions. We now discuss the experimental results.

7.1. Relative performance under different workload situations

Fig. 4 shows the sum of the aggregate real-time and fault-tolerance benefits accrued by BEN and HLC under the const/ramp load and ramp/ramp load patterns, respectively. We observe that the performance of BEN is close to that of HLC. Recall that HLC computes its allocation by considering replicas for *all* tasks at each step of the allocation process. BEN on the other hand, always selects that task for allocation which has the highest value for $(maxRTB_i + FTF_i)$ from the unallocated tasks. Thus, the close performance of BEN and HLC

Table 2
Parameters of the experimental task set

ID	Type	#Subtasks	Deadline (s)	$maxRTB_i$	FTF_i	$min.r_i$
0	Periodic	5	0.2	20	10	1
0	Aperiodic	5	0.2	25	10	1
1	Periodic	5	1.6	18	8	1
1	Aperiodic	5	1.6	23	8	1
2	Periodic	5	3.0	16	6	1
2	Aperiodic	5	3.0	21	6	1
3	Periodic	5	2.8	16	4	1
3	Aperiodic	5	2.8	19	4	1
4	Periodic	5	2.0	17	2	1
4	Aperiodic	5	2.0	17	2	1

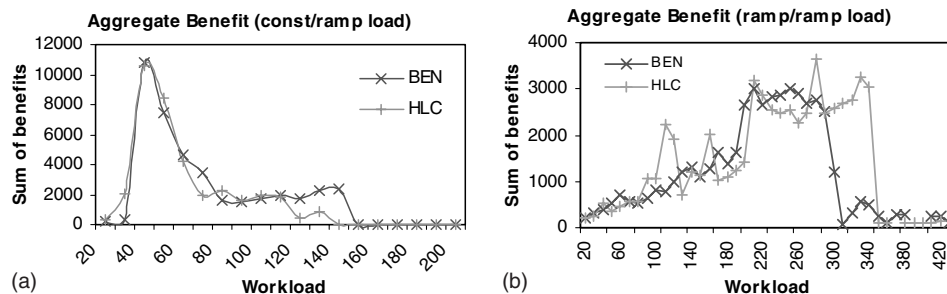


Fig. 4. Performance of BEN and HLC under increasing task workloads: (a) under const/ramp load and (b) under ramp/ramp load.

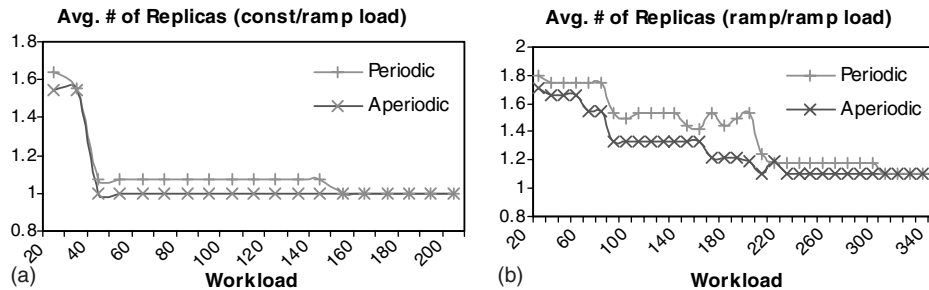


Fig. 5. Average number of replicas allocated by BEN: (a) under const/ramp load and (b) under ramp/ramp load.

suggests that allocating resources for the task with highest ($\max RTB_i + FTF_i$) significantly increases the sum of aggregate benefits. Replicating other tasks may also increase the sum of aggregate benefits, but not so significantly. Therefore, the BEN heuristic avoids searching a large solution space, but still reaps most of the benefit that HLC produces in much lower cost.

We also observe that the sum of the aggregate benefits shown in Fig. 4 starts with small values and continue increasing until the load is too heavy. Beyond such heavy load points, as the load further increases, the sum of aggregate benefits begins to decrease and eventually becomes very small.

We conjecture that when the load is too heavy, the algorithms determine that system resources are not sufficient for satisfying the task deadlines. At such heavy loads, the algorithms may find that the subtasks cannot meet their deadlines even if they are allocated the maximum number of replicas. It is also possible that the algorithms determine that the number of replicas needed to satisfy a subtask deadline is less than the maximum possible number of replicas, but close to the maximum. When the number of replicas is high, the messages generated by the replicas increase and can cause overload on the network. Thus, the algorithms allocate small number of replicas (possibly the minimum number of replicas), which cannot meet the task deadline, either.

To verify this intuition, we observe the average number of periodic subtask replicas and aperiodic subtask replicas allocated by BEN. The average number of subtask replicas are shown in Fig. 5. From the figure, we observe that beyond the heavy load points, the average number of replicas decreases while the load increases, confirming our intuition. The average number of replicas allocated by HLC was found to exhibit a similar pattern.

7.2. Relative performance under different failure situations

To study how BEN and HLC perform under different failure situations, we used the same (const/ramp) workload pattern for all tasks, but used the exponential failure-time model to determine the end-host reliability, and randomly generated end-host failures.

Fig. 6 shows the performance of BEN and HLC under three different failure patterns called 'A,' 'B,' and 'C.' The pattern 'A' considers a mean-time-to-failure (MTTF) of the three end-host groups to be 1000, 100, and 10 sec, respectively. Pattern 'B' considers an MTTF that is half of that of 'A,' and pattern 'C' considers an MTTF that is half of that of 'B.' Thus, the patterns constitute a failure rate of increasing intensity. From the figure, we observe that the performance of the algorithms degrade when the failure intensity increases. Furthermore, we observe that the performance of BEN is close to that of HLC.

7.3. Negotiation ability

We studied how the algorithms negotiate between real-time and fault-tolerance requirements. For example, if the fault-tolerance benefit factor of a task is larger than its real-time benefit, a "good" resource allocation algorithm should favor allocation decisions that yield higher fault-tolerance benefit but lower real-time benefit. The rationale for doing so is that when the fault-tolerance benefit factor increases (with respect to the real-time benefit), the loss of the real-time benefit may be compensated by the gain in the fault-tolerance benefit factor. This can potentially increase the sum of aggregate benefits.

Fig. 7(a) illustrates the performance of BEN when the fault-tolerance benefit factor is increased. The ratio of fault-tolerance benefit to the sum of aggregate benefits, denoted as FTB% in the figure, measures the fault-tolerance benefit accrued per unit total benefit accrued by the algorithm. The left bars in Fig. 7(a) show the ratio of fault-tolerance benefit to the sum of aggregate benefits and the ratio of real-time benefit to the sum of aggregate benefits produced by BEN under a baseline experiment, respectively. The right bars in Fig. 7(a) show the same ratio, but when the fault-tolerance benefit factor of all tasks is increased by a factor of 10 and the real-time benefit of all tasks is kept the same as that of the baseline experiments. From the right bars in Fig. 7(a), we observe that FTB% grows by approximately 10%.

Similarly, in Fig. 7(b), we show the effects of increasing the real-time benefits of tasks without increas-

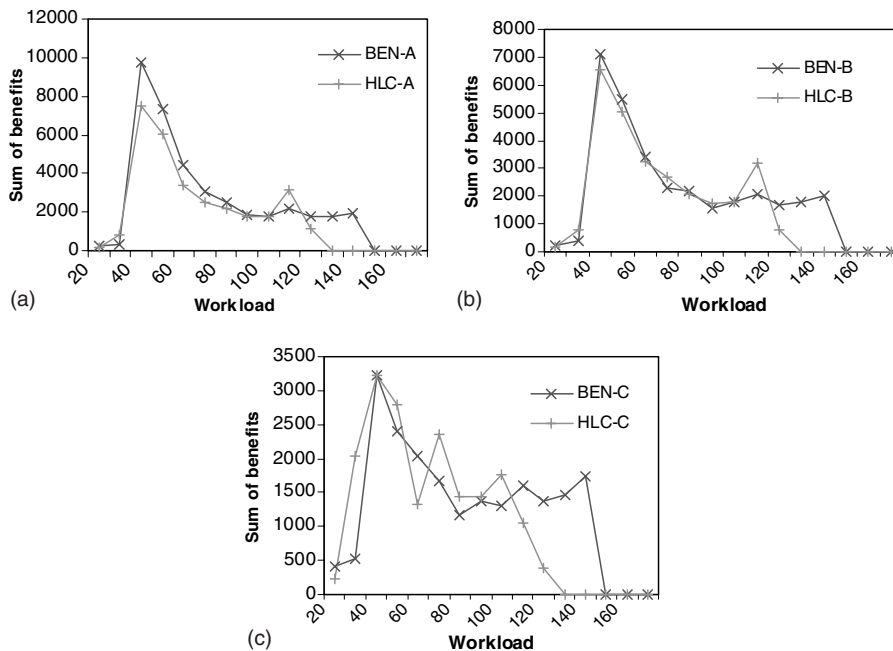


Fig. 6. Performance of BEN and HLC under processor failures: (a) failure pattern 'A'; (b) failure pattern 'B' and (c) failure pattern 'C'.

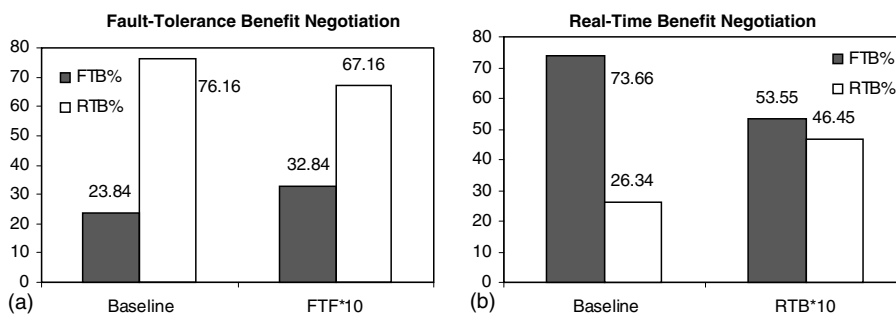


Fig. 7. Negotiation effectiveness of BEN: (a) $FTF > \max RTB$ and (b) $\max RTB > FTF$.

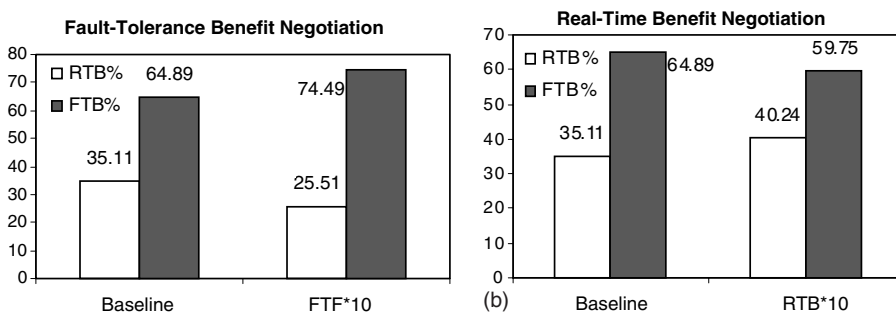


Fig. 8. Negotiation effectiveness of HLC: (a) $FTF > \max RTB$ and (b) $\max RTB > FTF$.

ing the fault-tolerance benefit factors. We observe that the negotiation effectiveness of BEN is even more significant here—the growth of RTB% is around 20%.

We conducted the same experiments for HLC. The results are shown in Fig. 8(a) and (b). We observe that for HLC, FTB% grows by around 10%, whereas RTB% grows by around 5%.

8. Conclusions

Our experimental results thus reveal that, in general, BEN's performance is quite close to that of HLC. Since BEN is faster than HLC by one order of magnitude, we regard this close performance of BEN to be quite significant.

We believe that BEN performs better because the algorithm only considers the currently highest $maxRTB_i + FTF_i$ task during an allocation, while HLC searches a much larger solution space. The experimental results confirm this conjecture that allocating resources to the highest $maxRTB_i + FTF_i$ task will significantly increase the system-level aggregate benefit. Allocating resources to other tasks may also increase the system aggregate benefit, but not that significantly.

We have constructed a real-time middleware called Choir (Ravindran, 2002b). Choir contains resource allocation algorithms that we had previously developed called RBA* and OBA (Hegazy and Ravindran, 2002b) and DRBA and DOBA (Hegazy and Ravindran, 2002a). We plan to implement BEN in Choir.

Acknowledgements

This paper is a revised and expanded version of our original paper with the same title, which appeared in the *Proceedings of ISCA 15th International Conference on Parallel and Distributed Computing Systems*, pp. 237–242, September 2002.

References

- Abdelzaher, T., Shin, K., 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., 1997. In: *Proceedings of The IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*. IEEE Computer Society, The 18th IEEE Real-Time Systems Symposium.
- Brandt, S., Nutt, G., Berk, T., Mankovich, J., 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., 1999. An adaptive, distributed airborne tracking system. In: *Proceedings of The Seventh IEEE International Workshop on Parallel and Distributed Real-Time Systems*. Lecture Notes in Computer Science, vol. 1586. Springer-Verlag, Berlin, pp. 353–362.
- Clark, R.K., 1990. Scheduling dependent real-time activities. Ph.D. Thesis, Carnegie Mellon University.
- DARPA, 1997. Quorum. Available from <http://www.darpa.mil/ipto/research/quorum/index.html>.
- Hegazy, T., 2001. Using application benefit for proactive resource allocation in asynchronous real-time distributed systems. Master's thesis, Virginia Polytechnic Institute and State University, USA.
- Hegazy, T., Ravindran, B., 2002a. On decentralized proactive resource allocation in asynchronous real-time distributed systems. In: *Proceedings of The Seventh IEEE/IEICE International Symposium on High Assurance Systems Engineering*, pp. 27–34.
- Hegazy, T., Ravindran, B., 2002b. Using application benefit for proactive resource allocation in asynchronous real-time distributed system. *IEEE Transactions on Computers* 51 (8), 945–962.
- Hermant, J.-F., Lann, G. L., 1998. A protocol and correctness proofs for real-time high-performance broadcast networks. In: *Proceedings of the Eighteenth International Conference on Distributed Computing Systems*, pp. 360–369.
- Jalote, P., 1994. *Fault-Tolerance in Distributed Systems*. Prentice-Hall, New Jersey.
- Jensen, E. D., 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, Berlin.
- Jensen, E.D., Ravindran, B., 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., 1997. Deadline assignment in a distributed soft real-time system. *IEEE Transactions on Parallel and Distributed Systems* 8 (12), 1268–1274.
- Koob, G., 1996. Quorum. In: *Proceedings of The Darpa ITO General PI Meeting*, pp. A-59–A-87.
- Lee, C., 1999. On quality of service management. Ph.D. Thesis, Carnegie Mellon University.
- Li, P., Ravindran, B., Hegazy, T., 2001. Implementation and evaluation of a best-effort scheduling algorithm in an embedded real-time system. In: *Proceedings of The IEEE International Symposium on Performance Analysis of Systems and Software*, Tucson, AZ, pp. 22–29.
- Mills, D.L., 1995. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networks* (June), 245–254.
- MK7.3a, 1998. MK7.3a Release Notes. The Open Group Research Institute's Real-Time Group, Cambridge, Massachusetts.
- Ravindran, B., 2001. Navy Collaborative Integrated Information Technology Initiative (NAVCIITI) Second Quarterly Report of Task 4.1 for Year 3. US Office of Naval Research.
- Ravindran, B., 2002a. Engineering dynamic real-time distributed systems: Architecture, system description language, and middleware. *IEEE Transactions on Software Engineering* 28 (1), 30–57.
- Ravindran, B., 2002b. Navy Collaborative Integrated Information Technology Initiative (NAVCIITI) Quarterly Report Number 21 of Task 4.1 for Year 4. Quarterly Progress Report Submitted to US Office of Naval Research.
- Rosu, D., Schwan, K., Yalamanchili, S., Jha, R., 1997. On adaptive resource allocation for complex real-time applications. In: *Proceedings of The 18th IEEE Real-Time Systems Symposium*, pp. 320–329.
- Stankovic, J.A., Spuri, M., Ramamritham, K., Buttazzo, G.C., 1998. *Deadline Scheduling for Real-Time Systems*. Kluwer Academic Publishers, Dordrecht.
- Varga, A., 2001. The omnet++ discrete event simulation system. In: *Proceedings of the European Simulation Multiconference (ESM'2001)*. Available from <http://whale.hit.bme.hu/omnetpp/>.
- Welch, L.R., Ravindran, B., Shirazi, B., Bruggeman, C., 1998. Specification and modeling of dynamic, distributed real-time systems. In: *Proceedings of The IEEE Real-Time Systems Symposium*, pp. 72–81.