

Scheduling Dependent Distributable Real-Time Threads in Dynamic Networked Embedded Systems

Sherif Fahmy, Binoy Ravindran
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{fahmy,binoy}@vt.edu

E. D. Jensen
The MITRE Corporation
Bedford, MA 01730, USA
jensen@mitre.org

Abstract

We consider scheduling distributable real-time threads with dependencies (e.g. due to synchronization) in partially synchronous systems in the presence of node failure. We present a distributed real-time scheduling algorithm called DQBUA. The algorithm uses quorum systems to coordinate nodes' activities when constructing a global schedule. DQBUA detects and resolves distributed deadlock in a timely manner and allows threads to access resources in order of their potential utility to the system. Our main contribution is handling resource dependencies using a distributed scheduling algorithm.

1 Introduction

Some emerging networked embedded systems are dynamic in the sense that they operate in environments with uncertain properties (e.g., [1]). These uncertainties include transient and sustained resource overloads (due to context-dependent activity execution times), arbitrary activity arrivals and completions, and arbitrary node failures and message losses. Reasoning about *end-to-end* timeliness is a difficult and unsolved problem in such systems. Another distinguishing feature of such systems is their relatively long activity execution time scales (e.g., milliseconds to minutes), which permits more time-costly real-time resource management.

Maintaining end-to-end properties (e.g., timeliness, connectivity) of a control or information flow requires a model of the flow's locus in space and time that can be reasoned about. Such a model facilitates reasoning about the contention for resources that occur along the flow's locus and resolving those contentions to seek optimal system-wide end-to-end timeliness. The *distributable thread* programming abstraction which first appeared in the Alpha OS [3], and later in the Real-Time CORBA 1.2 standard, directly provides such a model as their first-class programming and scheduling abstraction. A distributable thread is a single thread of execution with a globally unique identity that transparently extends and retracts through local and remote objects. We focus on distributable threads as our end-to-end programming abstraction, and hereafter, refer to them as *threads*, except as necessary for clarity.

Contributions. In this paper, we consider the problem of scheduling dependent threads in the presence of the previously mentioned uncertainties. Past efforts on thread scheduling (e.g., see [7]) and references therein) can be broadly categorized into two classes: *independent node scheduling* and *collaborative scheduling*. In the independent scheduling approach, threads are scheduled at nodes using propagated thread scheduling parameters and without any interaction with other nodes. Thread faults are managed by *integrity protocols* that run concurrent to thread execution. Integrity protocols employ failure detectors (or FDs), and use them to detect thread failures. In the collaborative scheduling approach, nodes explicitly cooperate to construct system-wide thread schedules, detecting node failures using FDs while doing so. We design a collaborative thread scheduling algorithm, DQBUA, that can handle dependencies. To the best of our knowledge, this is the first collaborative scheduling algorithm to consider dependencies. We compare DQBUA to RTG-DS [9], a dependent thread scheduling algorithm that uses gossip to improve the reliability of the communication layer and to find the next head node of a thread. RTG-DS falls under the independent category of thread scheduling algorithms.

2 Models and Objective

Distributable Thread Abstraction. Threads execute in local and remote objects by location-independent invocations and returns. A thread begins its execution by invoking an object operation. The object and the operation are specified when the thread is created. The portion of a thread executing an object operation is called a *thread segment*. Thus, a thread can be viewed as being composed of a concatenation of thread segments.

A thread's initial segment is called its *root* and its most recent segment is called its *head*. The head of a thread is the only segment that is active. A thread can also be viewed as being composed of a sequence of *sections*, where a section is a maximal length sequence of contiguous thread segments on a node. A section's first segment results from an invocation from another node, and its last segment performs a remote invocation.

Execution time estimates of the sections of a thread are assumed to be known when the thread arrives at the respective nodes. The time estimate includes that of the section's normal code, and can be violated at run-time (e.g., due to context dependence, causing processor overloads).

The sequence of remote invocations and returns made by a thread can typically be estimated by analyzing the thread code (e.g., [12]). The total number of sections of a thread is thus assumed to be known a-priori. The application is thus comprised of a set of threads, denoted $\mathbf{T} = \{T_1, T_2, \dots\}$. The set of sections of a thread T_i is denoted as $[S_1^i, S_2^i, \dots, S_k^i]$.

Timeliness Model. We specify the time constraint of each thread using a Time/Utility Function (TUF) [10]. A TUF allows us to decouple the urgency of a thread from its importance. This decoupling is a key property allowed by TUFs since the urgency of a thread may be orthogonal to its importance. A thread T_i 's TUF is denoted as $U_i(t)$. A classical deadline is unit-valued—i.e., $U_i(t) = \{0, 1\}$, since importance is not considered. Downward step TUFs generalize classical deadlines where $U_i(t) = \{0, \{m\}\}$. We focus on downward step TUFs, and denote the maximum, constant utility of a TUF $U_i(t)$, simply as U_i . Each TUF has an initial time I_i , which is the earliest time for which the TUF is defined, and a termination time X_i , which, for a downward step TUF, is its discontinuity point. $U_i(t) > 0, \forall t \in [I_i, X_i]$ and $U_i(t) = 0, \forall t \notin [I_i, X_i], \forall i$.

System Model. We consider a networked embedded system to consist of a set of client nodes $\pi^c = \{1, 2, \dots, N\}$ and a set of server nodes $\Pi = \{1, 2, \dots, n\}$ (*server* and *client* are logical designations given to nodes to describe the algorithm's behavior). Bi-directional logical communication channels are assumed to exist between every client-server and client-client pair. We also assume that these basic communication channels may lose messages with probability p , and communication delay is described by some probability distribution. On top of this basic communication channel, we consider a reliable communication protocol that delivers a message to its destination in probabilistically bounded time (with CDF $DELAY(t)$) provided that the sender and receiver both remain correct, using the standard technique of sequence numbers and retransmissions. We assume that each node is equipped with two processors; a processor that executes the sections hosted on the node and a scheduling coprocessor¹. We also assume that nodes in the system have access to GPS clocks that can provide each node to a UTC clock with nanosecond accuracy [5, 8, 13]. This is a realistic assumption in networked embedded systems such as Network Centric Warfare (NCW). We also assume that each node is equipped with QoS failure detectors (FDs) [2]. The QoS FD described in [2] is designed to monitor only one process. Therefore, we equip each node with $N - 1$ FDs to monitor the status of all other nodes. On each node, i , these $N - 1$ FDs output the nodes they suspect to the the same list, $suspect_i$.

Exceptions and Abort Model. Each section of a thread has an associated exception handler. We consider a termination model for thread failures including time-constraint violations and node failures.

If a thread has not completed by its termination time, a time constraint-violation exception is raised by DQBUA, and handlers are released on all nodes that host the thread's sections by DQBUA. When a handler executes (not necessarily when it is released), it will abort the associated section after performing compensations and recovery actions that are necessary to avoid inconsistencies—e.g., rolling back/forward, or making other compensations to logical and physical resources that are held by the section to safe states.

We consider a similar abort model for node failures. When a thread encounters a node failure causing orphans, DQBUA delivers failure-exception notifications to all orphan nodes of the thread. Those nodes then respond by releasing handlers which abort the orphans after executing compensating actions.

Each handler may have a time constraint, which is specified as a relative deadline. Each handler also has an execution time estimate. This estimate along with the handler's deadline are described by the handler's thread when the thread arrives at a node. Violation of the termination time of a handler's deadline will cause the immediate execution of system recovery code on that node, which will recover the thread section's held resources and return the system to a consistent and safe state.

¹Dual core technology makes this assumption realistic.

Failure Model. The nodes are subject to crash failures. When a process crashes, it loses its state memory — i.e., there is no persistent storage. If a crashed client node recovers at a later time, we consider it a new node since it has already lost all of its former execution context. A client node is *correct* if it does not crash; it is *faulty* if it is not correct. In the case of a server crash, it may either recover or be replaced by a new server assuming the same server name (using DNS or DHT — e.g. [6] — technology). We model both cases as server recovery. Since crashes are associated with memory loss, recovered servers start from their initial state. A server is *correct* if it does not fail; it is *faulty* if it is not correct. DQBUA tolerates up to $N - 1$ client failures and up to $f_{max}^s \leq n/3$ server failures (see [7]). The actual number of server failures is denoted as $f^s \leq f_{max}^s$ and the actual number of client failures is denoted as $f \leq f_{max}$ where $f_{max} \leq N - 1$.

Resource Model Threads can access serially reusable non-CPU resources (e.g., disks, NICs) located at their nodes during their execution. We consider the single resource model where only one instance of each resource exists in the system. Resources can be shared under mutual exclusion constraints. A thread may request multiple resources during its lifetime but can only have one outstanding request at any given instance of time. Threads explicitly release all granted requests before the end of their execution.

All resource request/release pairs are assumed to be confined within one node. Thus, a node cannot lock a resource on one node and release it on another. However, it is possible for a thread to lock a resource on a node and then make a remote invocation to another node (carrying the lock with it). Since resource request/release pairs are confined to one node, the lock is released when the thread’s node returns back to the node on which the resource was acquired.

Resources are assumed to access their resources in an arbitrary order — i.e., which resources are needed by which threads is not known a priori. Consequently we employ deadlock detection and resolution methods instead of prevention and avoidance techniques. Deadlock resolution is performed by aborting one of the deadlocked threads by executing its exception handler.

Scheduling Objectives. Our primary objective is to design a thread scheduling algorithm that will maximize the total utility accrued by all threads as much as possible in the presence of dependencies. Further, the algorithm must provide assurances on the satisfaction of thread termination times in the presence of (up to f_{max}) crash failures. Moreover, the algorithm must bound the time threads remain in deadlock.

3 Algorithm Rationale

In [7], we develop QBUA, a scheduling algorithm for distributable real-time threads in partially synchronous systems. In this work, we extend QBUA by adding resource dependencies and precedence constraints handling capabilities, we call the resulting algorithm DQBUA. As in [4], precedence constraints can be programmed as resource dependencies and are handled the same way.

As in QBUA, when a node detects a distributed scheduling event (the failure of a node, the arrival of a new thread into the system or a resource request) it contacts a quorum system requesting permission to run an instance of DQBUA (in order to construct a global schedule). All other scheduling events, such as resource releases and section completion, are dealt with locally, see Algorithm 1. Once permission is granted, it broadcasts a start of algorithm message to all other nodes requesting their scheduling information. Nodes that receive this message reply by sending their scheduling information. When all nodes have sent their scheduling information to the requesting node, it computes a system-wide schedule, which we call a System Wide Executable Thread Set (or SWETS), and multicasts any updates to nodes whose schedule has been affected.

The purpose of the quorum system is to arbitrate among nodes that detect a distributed scheduling event concurrently. This arbitration reduces thrashing by minimizing the number of instances of DQBUA that are started to handle the same or concurrent scheduling events. Due to space limitations, we do not reproduce the details of the quorum arbitration algorithm, see [7] for details.

While computing a system-wide schedule, threads are ordered in non-increasing order of their global Potential Utility Density (PUD) (which we define as the ratio of a thread’s utility to its remaining execution time), the threads are then considered for scheduling in that order. Favoring high global PUD threads allows us to select threads for scheduling that result in the most increase in system utility for the least effort. This heuristic attempts to maximize total accrued utility [4].

Both local and distributed resource dependencies are possible, therefore both local and distributed deadlock can occur. By considering resource requests as distributed scheduling events, DQBUA detects and resolves both local and distributed deadlock in a timely manner. In addition, contention for resources is resolved using their global PUD.

Algorithm 1: Event Dispatcher on each node i

```
1 Data: schedevent, current schedule  $\sigma_p$ ;  
2 switch schedevent do  
3   case invocation arrives for  $S_j^i$   
4     | mark segment  $S_j^i$  ready;  
5   case segment  $S_j^i$  completes  
6     | remove  $S_j^i$  from  $\sigma_p$ ;  
7     | remove  $S_j^h$  from  $H$ ;  
8     | set  $RE_j^i$  to zero;  
9   case  $S_j^h \in H$  and  $S_i^h$  .st expires  
10    | mark handler  $S_j^h$  ready;  
11  case downstream handler  $S_{j+1}^h$  completes  
12    | mark handler  $S_j^h$  ready;  
13  case handler  $S_j^h$  completes  
14    | remove  $S_j^h$  from  $\sigma_p, H$ ;  
15    | notify scheduler for  $S_{j-1}^h$ ;  
16  case new thread,  $T_i$ , arrives  
17    | if origin node, send segments  $S_j^i$  to all;  
18    | pass event to DQBUA;  
19  case node failure detected  
20    | pass event to DQBUA;  
21  case  $S_j^i$  requests a resource  
22    | pass event to DQBUA;  
23  case  $S_j^i$  releases a resource  
24    | free resource;  
25 execute first ready segment in  $\sigma_p$ ;
```

4 Algorithm Description

Once the arbitration phase of the algorithm is complete and a node has been granted permission to run an instance of DQBUA, that node runs the algorithm depicted in Algorithm 2. In Algorithm 2, the node first broadcasts a start of algorithm message (line 1) and then waits $2T$ time units for all nodes in the system to respond by sending their local scheduling information (line 2). After collecting this information, the node computes SWETS (line 3) using Algorithm 5. After computing SWETS, the node contacts affected nodes (i.e. nodes that will have sections added or removed from their schedule).

Algorithm 2: Compute SWETS

```
1: Broadcast start of algorithm message, START;  
2: Wait  $2T$  collecting replies from other nodes;  
3: Construct SWETS using information collected;  
4: Multicast change of schedule to affected nodes;  
5: return;
```

Algorithm 5 is used by a client node to compute SWETS once it has received information from all other nodes in the system (line 3 in Algorithm 2). It performs two basic functions, first, it computes a system wide order on threads by computing their global PUD. It then attempts to insert the remaining sections of each thread, in non-increasing order of global PUD, into the scheduling queues of all nodes in the system. After the insertion of each thread, the schedule is checked for feasibility. If it is not feasible, then the thread is removed from SWETS (after scheduling the appropriate exception handler if necessary).

First we need to define the global PUD of a thread. Assume that a thread, T_i , has k sections denoted $\{S_1^i, S_2^i, \dots, S_k^i\}$. We define the global remaining execution time, GE_i , of the thread to be the sum of the remaining execution times of each of the thread's sections. Let $\{RE_1^i, RE_2^i, \dots, RE_k^i\}$ be the set of remaining execution times of T_i 's sections, then $GE_i = \sum_{j=1}^k RE_j^i$. Assuming that we are using step-down TUFs, and T_i 's TUF is $U_i(t)$, then its global PUD can be computed as: $T_i.PUD = U_i(t_{curr} + GE_i)/GE_i$, where U is the utility of the thread and t_{curr} is the current time. Using global PUD, we can establish a system wide order on the threads in non-increasing order of "return on investment". Thus we consider the threads for scheduling in an order that is designed to maximize accrued utility [4].

In the absence of dependencies, the above computation can be used to represent the utility that would be accrued if a

thread where to execute immediately. However, since we consider dependencies, the utility of a thread can only be accrued if all the threads it depends on either complete their execution or are aborted first. Therefore when a section requests a resource, we compute its dependency list by following the chain of resource requests and ownership. Since a resource request is a distributed scheduling event, the node that gets permission to run an instance of DQBUA (after arbitration by the quorum system) will be sent all the information necessary for it to compute the dependency chain.

Once the dependency list has been computed, we compute the PUD of the current thread by using a least effort heuristic —i.e., while examining the threads in the dependency list to compute PUD, if it is faster to abort them than to continue execution, then the threads are aborted and vice versa. Thus we compute the PUD of a thread if it is executed as soon as possible. A similar heuristic is used in [4] but for a single processor, in contrast to the distributed system we consider in this work. Note that this heuristic minimizes the amount of time a high utility thread waits for a resource, at the expense of having to possibly re-execute threads that have been aborted (see [4] for details).

Algorithm 3: computePUD

```

1: Input:  $T_i, Dep(i, k), j$ ; //  $j$  is node where resource request occurred
2:  $Util \leftarrow 0; Time \leftarrow 0; Seen \leftarrow \emptyset;$ 
3: for each  $Dep(i, k)$  do
4:   for each  $S \in Dep(i, k)$  do
5:     if  $S.ID \notin Seen$  then
6:        $Seen \leftarrow Seen \cup S.ID;$ 
7:       //  $\Gamma_1$ : sections from  $S$  until last visit to  $j$ 
8:        $S.Rem \leftarrow \sum_{k \in \Gamma_1} RE_k^{S.ID};$ 
9:       //  $\Gamma_2$ : all downstream sections
10:       $S.Abort \leftarrow \sum_{k \in \Gamma_2} S_k^h.ex;$ 
11:      if  $S.Abort > S.Rem$  then
12:         $Time \leftarrow Time + S.Rem;$ 
13:         $Util \leftarrow Util + U_T(t_{curr} + S.Rem)$ 
14:      else  $Time \leftarrow Time + S.Abort;$ 
15:  $Time \leftarrow Time + GE_i;$ 
16:  $Util \leftarrow Util + U_i(t_{curr} + GE_i);$ 
17:  $T_i.PUD = Util/Time;$ 
18: return  $T_i.PUD;$ 

```

Since we are computing the PUD of the whole thread, we need to consider the dependencies of all sections belonging to the thread. Therefore, Algorithm 3 considers the dependency list, $Dep(i, k)$, of each of the k sections of thread T_i while computing the PUD. While computing the global PUD of a thread, we take into account the utility of the threads that it depends on. The reason that we do this is that in order to schedule a thread, we need to schedule its dependencies first, so when the thread completes, its dependencies will also have completed thus accruing the utility of both the dependencies and the thread itself. Thus we compute the utility of completing the current thread as the sum of the utility of the current thread and the threads it depends on (lines 13 and 16). We measure the potential utility of a thread and its dependents as the ratio of the utility they can accrue and the time taken for them to accrue this utility (line 17).

We assume that each thread in the system has a globally unique ID, and that each of the thread's sections, S_i , store this global ID in the variable, $S_i.ID$. Since a thread has multiple sections, each of these sections may be dependent on a number of different sections. It is possible that two sections of a thread are dependent on two sections of another thread. In this case, we should only consider the utility of the dependent thread once (since the utility of this thread will only be accrued once when it completes execution). Therefore, in line 5, we check whether a section in a dependency chain, $Dep(i, k)$, belongs to a thread that has been handled before (because another of its sections is in the dependency list of a different section belonging to the current thread). Only threads that have not been considered before are used to compute the current thread's global PUD.

Note that when computing the time remaining for a section, S , to release a resource (line 8), we consider the remaining time for sections starting from S until the last section belonging to S 's thread arrives at the current node j . The reason for this is that we do not know when the resource will be released by section S , however since we assume that all resource request/release pairs occur on the same node (see Section 2), the latest time at which the resource will be released is when the last section belonging to S 's thread to visit node j terminates.

Similarly, when we compute the abort time for section S (line 10), we only consider the abort times of downstream sections. The reason for this is that DQBUA ensures LIFO execution of abort handlers and therefore the current section's abort handler will only execute after the handlers of its downstream sections have terminated. Note that neither the abort time computed in line 10 nor the remaining time to release the resource computed in line 8 are actual times at which those events

will occur (interference by other threads ensures that this is not the case), rather the values are merely used as an indication of the amount of work necessary to abort a section to release its resources or to complete a section to release its resources respectively. Since we consider a heuristic of performing the least amount of work, we choose the scenario that takes the least amount of time (line 13).

When computing the global PUD of a section, we need to have up-to-date information about threads that have a section in $Dep(i, k)$. Since resource requests are distributed scheduling events, this information will be received when all nodes in the system send their scheduling information to the node constructing the schedule in response to its broadcast start of algorithm message (lines 1-2 in Algorithm 2).

We now turn our attention to the method used to check schedule feasibility. For a schedule to be feasible, all the sections it contains should complete their execution before their assigned termination time. Since we are considering threads with end-to-end termination times, the termination time of each section needs to be derived from its thread's end-to-end termination time. This derivation should ensure that if all the section termination times are met, then the end-to-end termination time of the thread will also be met. For the last section in a thread, we derive its termination time as simply the termination time of the entire thread. The termination time of the other sections is the latest start time of the section's successor minus the communication delay. Thus the section termination times of a thread T_i , with k sections, is:

$$S_j^i.tt = \begin{cases} T_i.tt & j = k \\ S_{j+1}^i.tt - S_{j+1}^i.ex - T & 1 \leq j \leq k - 1 \end{cases}$$

where $S_j^i.tt$ denotes section S_j^i 's termination time, $T_i.tt$ denotes T_i 's termination time, and $S_j^i.ex$ denotes the estimated execution time of section S_j^i . The communication delay, which we denote by T above, is a random variable Δ . Therefore, the value of T can only be determined probabilistically. This implies that if each section meets the termination times computed above, the whole thread will meet its termination time with a certain, high, probability. In addition, each section's handler has a **relative** termination time, $S_j^h.X$. However, a handler's **absolute** termination time is relative to the time it is released, more specifically, the **absolute** termination time of a handler is equal to the sum of the **relative** termination time of the handler and the failure time t_f (which cannot be known a priori). To overcome this problem, we delay the execution of the handler as much as possible, thus leaving room for more important threads. We compute the handler termination times as follows:

$$S_j^h.tt = \begin{cases} S_k^i.tt + S_j^h.X + T_D + t_a & j = k \\ S_{j+1}^h.tt + S_j^h.X + T & 1 \leq j \leq k - 1 \end{cases}$$

where $S_j^h.tt$ denotes section handler S_j^h 's termination time, $S_j^h.X$ denotes the relative termination time of section handler S_j^h , $S_k^i.tt$ is the termination time of thread i 's last section, t_a is a correction factor corresponding to the execution time of the scheduling algorithm, and T_D is the time needed to detect a failure by our QoS FD [2]. From this, we compute latest start times for each handler: $S_j^h.st = S_j^h.tt - S_j^h.ex$ for $1 \leq j \leq k$, where $S_j^h.ex$ denotes the estimated execution time of section handler S_j^h . Using these derived termination times, we can check whether a schedule is feasible or not.

Algorithm 4 shows how this is done in DQBUA. If the estimated completion time, $S_i.Fin$, of a section is greater than its derived termination, $S_i.tt$, then the schedule is not feasible (lines 13-14). We compute $S_i.Fin$ as the sum of the start time of a section and its execution time. However, it is important to note that, except for current and previous head nodes, these sections haven't arrived at their nodes yet when Algorithm 4 is invoked. Therefore we need to estimate the start time of these sections when computing the estimated completion time of those sections.

We estimate the start time of a section to be the maximum of the estimated completion time of the section preceding it in the local queue (line 10) and the arrival time of the section on a node (which we estimate as the sum of the completion time of the section's predecessor and the communication delay, $S_{i-1}.Fin + T$). We assume that each section's estimated completion time, $S_i.Fin$, is set to infinity before algorithm Algorithm 4 is run.

We use this relatively expensive method for checking the feasibility of schedules since alternative methods can be misleading. As mentioned before, when Algorithm 4 is invoked, only the current head section and its predecessors will have started in the system, thus the start time of the remaining sections in a thread need to be estimated. The expedient method, used in some previous work, of using the latest start time of a section (computed as its predecessor's latest termination time plus a communication delay, $S_{i-1}.tt + T$) as an estimated for a section's start time means that a section will have no slack time in the schedule. Therefore, the section cannot tolerate any interference by other sections. This usually leads to pessimistic results with some threads being rejected from an underloaded system. Algorithm 4 handles this by computing a better estimate of the start time of sections that haven't started yet.

Algorithm 4: isFeasible

```
1: Input:  $\sigma_i$ ; //Schedule for each node
2: for  $1 \leq i \leq N$  do
3:    $pos_i \leftarrow 1$ ;
4: Until ( $pos_i = \text{length}(\sigma_i)$ ,  $1 \leq i \leq N$ ) do
5:   for  $1 \leq i \leq N$  do
6:      $S_i \leftarrow \text{getElement}(\sigma_i, pos_i)$ ;
7:      $pre \leftarrow \text{getElement}(\sigma_i, pos_i - 1)$ ;
8:     if  $pos_i = 1$  then  $pre.Fin \leftarrow 0$ ;
9:     if  $i = 1$  then  $S_{i-1}.Fin \leftarrow S_i.Arr$ ;  $T \leftarrow 0$ ;
10:     $Start \leftarrow \max(pre.Fin, S_{i-1}.Fin + T)$ ;
11:    if  $Start \neq \infty$  then
12:       $S_i.Fin \leftarrow S_i.ex + Start$ ;
13:      if  $S_i.Fin > S_i.tt$  then
14:         $\leftarrow \text{return } false$ ;
15:       $pos_i \leftarrow pos_i + 1$ ;
16: return true;
```

Algorithm 5: ConstructSchedule

```
1: input:  $\Gamma$ ; //Set of threads in the system
2: input:  $\sigma_j^p, H_j \leftarrow \text{nil}$ ; // $\sigma_j^p$ : Previous schedule of node  $j$ ,  $H_j$ : set of handlers scheduled
3: for each  $T_i \in \Gamma$  do
4:   if for some section  $S_j^i \in T_i$ ,  $t_{curr} + S_j^i.ex > S_j^i.tt$  then  $T_i.PUD \leftarrow 0$ ;
5:   else
6:      $\leftarrow \text{Compute } Dep(i, j)$ , resolving deadlock if necessary;
7:      $T_i.PUD \leftarrow \text{ComputePUD}(T_i, Dep(i, j))$ ;
8: for each task  $el \in \sigma_j^p$  do
9:    $\leftarrow \text{if } el \text{ is an exception handler for section } S_j^i \text{ then Insert}(el, H_j, el.tt)$ ;
10:  $\sigma_j \leftarrow H_j$ ;
11:  $\sigma_{temp} \leftarrow \text{sortByPUD}(\Gamma)$ ;
12: for each  $T_i \in \sigma_{temp}$  do
13:    $T_i.stop \leftarrow false$ ;
14:   if do not receive  $\sigma_j$  from node hosting  $S_j^i \in T_i$  then
15:      $\leftarrow T_i.stop \leftarrow true$ ;
16:   if  $T_i.PUD > 0$  and  $T_i.stop \neq true$  then
17:      $\leftarrow \text{insertByEDF}(T_i, Dep(i, j))$ ;
18: for each  $j \in N$  do
19:    $\leftarrow \text{if } \sigma_j \neq \sigma_j^p \text{ then Mark node } j \text{ as being affected}$ ;
```

In Algorithm 5, each node, j , sends the node running DQBUA its current local schedule σ_j^p . Using these schedules, the node can determine the set of threads, Γ , that are currently in the system. Both these variables are inputs to the scheduling algorithm (lines 1 and 2 in Algorithm 5). In lines 3-8, the algorithm DQBUA computes the global PUD of each thread in Γ . The global PUD is computed by first checking whether all sections in a thread can complete execution before their termination time if they were executed immediately. If this is not the case, the thread is assigned a PUD of zero since it cannot possibly accrue any utility to the system (lines 4). Otherwise, we compute the dependency chain for the thread's sections and call Algorithm 3 to compute the global PUD of the thread (lines 6-7). In line 6, we check for cycles to detect any deadlock that may exist. If a cycle is found, it is broken by aborting the thread with the least PUD by executing its exception handler.

Before we schedule the threads, we need to ensure that the exception handlers of any thread that has already been accepted into the system can execute to completion before its termination time. We do this by inserting the handlers of sections that were part of each node's previous schedule into that node's current schedule (lines 8-9). Since these handlers were part of σ_j^p , and DQBUA always maintains the feasibility of a schedule as an algorithm invariant, we are sure that these handlers will meet their termination times.

Algorithm 6: insertByEDF

```

1: input:  $\sigma_j^p, \sigma_j$ ;
2:  $\sigma_j^{tmp} \leftarrow \sigma_j$ ; // make a copy of the schedule
3: for each remaining section,  $S_j^i$ , belonging to  $T_i$  do
4:   if  $S_j^i \notin \sigma_j^{tmp}$  then
5:     Insert( $S_j^i, \sigma_j^{tmp}, S_j^i.tt$ );
6:      $TT_{cur} \leftarrow S_j^i.tt$ ;
7:     if  $S_j^h \notin \sigma_j^p$  then Insert( $S_j^h, \sigma_j^{tmp}, S_j^h.tt$ );
8:     for  $\forall S_n^k \in Dep(i, j)$  do
9:       if  $S_n^k \in \sigma_n^{tmp}$  then
10:        if  $S_n^k$  is an abortion handler then
11:          Remove all sections belonging to  $S_n^k$ 's thread;
12:           $TT \leftarrow \text{lookUp}(S_n^k, \sigma_n^{tmp})$ ;
13:          if  $TT < TT_{cur}$  then
14:             $TT_{cur} \leftarrow TT$ ;
15:            Continue;
16:          else
17:            Remove( $S_n^k, \sigma_n^{tmp}, TT$ );
18:            Insert( $S_n^k, \sigma_n^{tmp}, TT_{cur}$ );
19:             $\delta \leftarrow TT - TT_{cur}$ ;
20:            for all predecessors,  $S_i^x$ , of  $S_n^k$  do
21:              //If  $S_n^k$  is an abortion handler,  $S_i^x$ s are also abortion handlers.
22:              //Otherwise,  $S_i^x$ s are normal sections
23:               $TT \leftarrow \text{lookUp}(S_i^x, \sigma_i^{tmp})$ ;  $\gamma \leftarrow \delta$ ;
24:              if  $S_n^k.tt - TT < \delta$  then
25:                 $\gamma \leftarrow \delta - (S_n^k.tt - TT)$ ;
26:              Remove( $S_i^x, \sigma_i^{tmp}, TT$ );
27:              Insert( $S_i^x, \sigma_i^{tmp}, TT - \gamma$ );
28:            else
29:               $TT_{cur} \leftarrow \min(TT_{cur}, S_n^k.tt)$ ;
30:              Insert( $S_n^k, \sigma_n^{tmp}, TT_{cur}$ );
31:              if  $S_n^k$  is not an abortion handler then
32:                if  $S_n^h \notin \sigma_n^p$  then Insert( $S_n^h, \sigma_n^{tmp}, S_n^h.tt$ );
33: if isFeasible( $\sigma_j^{tmp}, s$ )=true then
34:    $\sigma_j \leftarrow \sigma_j^{tmp}$  for all  $j$ ;
35: return  $\sigma_j$  for all  $j$ ;

```

In line 11, we sort the threads in the system in non-increasing order of PUD and consider them for scheduling in that order (lines 12-17). In lines 14-15 we mark as failed any thread that has a section hosted on a node that does not participate in the algorithm. If a thread can contribute non-zero utility to the system and the thread has not been rejected from the system, then we insert its sections, and their dependencies, into the scheduling queue of the node responsible for them in non-decreasing order of termination time by calling Algorithm 6 (lines 16-17).

When Algorithm 6 is invoked, a copy is made of the current schedule so that any changes that result in an infeasible schedule can be undone (line 2). We then consider each of the remaining sections of the thread being considered, if the section does not already belong to the current schedule (because it was part of the dependency chain of a previous thread), the section and its handler are inserted into the current schedule (lines 5-7).

We then consider the dependencies of that section (lines 8-32). Although sections are considered for scheduling in non-increasing order of global PUD, they are inserted into the schedule in non-decreasing termination time order. Thus during underloads, when no threads are rejected, the resulting schedule is basically a deadline ordered list. So during underloads, our scheduling algorithm defaults to Earliest Deadline First (EDF) scheduling, which is an optimal realtime scheduling algorithm [11] that accrues 100% utility during underloads. Note that if a section, S_n^k , in the dependency chain, $Dep(i, j)$, needs to be aborted in order to reduce the blocking time of a thread, then all the sections belonging to S_n^k 's thread need to be aborted as well (lines 10-11).

In order to ensure that the order of the dependencies is maintained, if the termination time of a section is greater than the termination time of a section that depends on it, its termination time is moved up to the termination time of the section that depends on it (lines 17 and 27). In addition, all the predecessors of that current section have their termination time adjusted to reflect this new value (lines 20-27).

5 Algorithm Properties

We now turn our attention to proving some theoretical results for the algorithm. Below, T is the communication delay, Γ is the set of threads in the system and k is the maximum number of sections in a thread.

Lemma 1. *A node determines whether or not it needs to run an instance of DQBUA at most $4T$ time units after it detects a distributed scheduling event, with high, computable probability, P_{lock} .*

Proof. This follows from Lemma 1 in [7]. □

Lemma 2. *Once a node is granted permission to run an instance of DQBUA, it takes $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units to compute a new schedule, with high, computable, probability, P_{SWETS} .*

Proof. Once a node is granted permission to run an instance of DQBUA it executes Algorithm 2. This algorithm has three communication steps, one to broadcast the START message, another to receive the replies from other nodes in the system and one to multicast any changes to affected nodes. Thus the algorithm takes a total of $3T$ time units for its communication with other nodes. In addition to these communication steps, Algorithm 2 also takes time to actually compute SWETS (line 3). Algorithm 5 is the algorithm that is used to compute SWETS. In this algorithm, lines 3-7 iterate $|\Gamma|$ times, and the function computePUD, invoked in line 7, takes $|\Gamma|k^2$ time in the worst case. Therefore the time complexity of lines 3-7 is $|\Gamma|^2 k^2$.

Lines 8-9 take $O(|\Gamma|k)$ time in the worst case, line 11 sorts the threads in $O(|\Gamma| \log(|\Gamma|))$ time. The for loop in lines 12-17 iterates $|\Gamma|$ times in the worst case. The body of this loop calls Algorithm 6 in line 17. The time complexity of Algorithm 6 is dominated by three nested loops in lines 3-32. The outer loop iterates k times in the worst case, the middle loop (starting at line 8) iterates $O(|\Gamma|k)$ times in the worst case, and the inner most loop (starting at line 20) iterates k times in the worst case. The time complexity of the body of the inner loop is dominated by the time complexity of the peek, insert and remove operations (lines 23, 26 and 27 respectively). Using self-balancing binary search trees to represent the queues, all these operations can be performed in $O(\log(|\Gamma|k))$ time. Therefore the nested loop structure, and thus Algorithm 6, has a time complexity of $O(|\Gamma|k^3 \log(|\Gamma|k))$. Therefore the time complexity of lines 12-17 is $O(|\Gamma|^2 k^3 \log(|\Gamma|k))$.

Thus the complexity of the algorithm is $O(|\Gamma|^2 k^2 + |\Gamma|k + |\Gamma| \log(|\Gamma|) + |\Gamma|^2 k^3 \log(|\Gamma|k))$, which is asymptotically $O(|\Gamma|^2 k^3 \log(|\Gamma|k))$. Adding the communication delay to this computational complexity we get $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + 3T)$, which is asymptotically $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$. There are three communication rounds in this procedure. However, the first two of these communication rounds depend on timeouts (line 2, Algorithm 2), therefore it is only the third that is probabilistic in nature. Therefore, the probability that SWETS is computed in the time derived above is equal to the probability that the nodes receive the multicast message sent on line 4 of Algorithm 2 within T time units. Since the communication delay has CDF $DELAY(t)$, the probability that T is not violated during runtime, and thus that the time bound above is respected, is $P_{SWETS} = DELAY(T)$. □

Theorem 3. *A distributed scheduling event is handled at most $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T + T_D)$ time units after it occurs, with high, computable, probability, P_{hand} .*

Proof. There are three possible distributed scheduling events: 1) the arrival of a new thread into the system, 2) a resource request and 3) the failure of a node.

In case of the arrival of a new thread or a resource request, the head node of the thread immediately attempts to acquire a “lock” on running an instance of DQBUA. By Lemma 1, the node takes $4T$ time units to acquire a lock and by Lemma 2, it takes the algorithm $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ to compute SWETS. Therefore, in the case of the arrival of a thread or a resource request the event is handled $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T + 4T) = O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units after it occurs. Note that $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ is $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T + T_D)$.

In case of a node failure, some node will detect this failure after T_D time units. That node then attempts to acquire a lock from the quorum system to run an instance of DQBUA. By Lemmas 1 and 2, this takes $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units. Thus the event is handled $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T + T_D)$ time units after it occurs.

In both these cases, the result relies on Lemmas 1 and 2, so the probability that events are handled within the time frame mentioned above is $P_{hand} = P_{SWETS} \times P_{lock}$. \square

Lemma 4. *If all nodes are underloaded and no nodes fail, then no threads will be suggested for rejection by DQBUA with high, computable, probability p_{norej} .*

Proof. Since the nodes are all underloaded and no nodes fail, Algorithm 5 ensures that all sections will be accepted for scheduling in the system. Therefore, the only source of thread rejection is if a node does not receive a suggestion from other nodes during the timeout value, $2T$, (see line 2 in Algorithm 2). This can occur due to one of two reasons; 1) the broadcast message (line 1, Algorithm 2), that indicates the start of the algorithm, may not reach some nodes 2) the broadcast message reaches all nodes, but these nodes do not send their suggestions to the node running DQBUA during the timeout value assigned to them.

The probability that a node does not receive a message within the timeout value from one of the other nodes is $p = 1 - DELAY(T)$. We consider the broadcast message to be a series of unicasts to all other nodes in the system. Therefore, the probability that the broadcast START message reaches all nodes is $P_{tmp} = \text{bino}(0, N, p)$ where $\text{bino}(x, n, p)$ is the binomial distribution with parameters n and p . If the START message is received, each node sends its schedule to the node that sent the START message. The probability that none of these messages violate the timeout is $tmp = \text{bino}(0, N, p)$. As mentioned before, if none of the nodes miss a message, no threads will be rejected, thus the probability that no threads will be rejected is the product of the probability that the broadcast message reaches all nodes, and the probability that all nodes send their schedule before the timeout expires. Therefore, $p_{norej} = tmp \times P_{tmp}$. \square

Lemma 5. *If each section of a thread meets its derived termination time, then under DQBUA, the entire thread meets its termination time with high, computable probability, p_{suc} .*

Proof. Since the termination times derived for sections are a function of communication delay and this communication delay is a random variable with CDF $DELAY(t)$ the fact that all sections meet their termination times implies that the whole thread will meet its global termination time only if none of the communication delays used in the derivation are violated during runtime.

Let T be the communication delay used in the derivation of section termination times. The probability that T is violated during runtime is $p = 1 - DELAY(T)$. For a thread with k sections, the probability that none of the section to section transitions incur a communication delay above T is $p_{suc} = \text{bino}(0, k, p)$. Therefore, the probability that the thread meets its termination time is also $p_{suc} = \text{bino}(0, k, p)$. \square

Theorem 6. *If all nodes are underloaded, no nodes fail (i.e. $f = 0$) and each thread can be delayed $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units once and still be schedulable, DQBUA meets all the thread termination times yielding optimal total utility with high, computable, probability, P_{alg} .*

Proof. By Lemma 4, no threads will be considered for rejection from a fault free, underloaded system with probability p_{norej} . This means that all sections will be scheduled to meet their derived termination times by Algorithm 5.

By Lemma 5, this implies that each thread, j , will meet its termination time with probability p_{suc}^j . Therefore, for a system with $X = |\Gamma|$ threads, the probability that all threads meet their termination time is $P_{tmp} = \prod_{j=1}^X p_{suc}^j$. Given that the probability that all threads will be accepted is p_{norej} , $P_{alg} = P_{tmp} \times p_{norej}$.

We make the requirement that a thread tolerate a delay of $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units and still be schedulable because DQBUA takes $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units to reach its decision about the schedulability of a newly arrived thread. Thus if this delay causes any of the thread’s sections to miss their deadlines, the thread will not be schedulable. We

only require that the thread suffer this delay *once* because we assume that there is a scheduling coprocessor on each node, thus the delay will only be incurred by the newly arrived thread while other threads continue to execute uninterrupted on the other processor. \square

Theorem 7. *If $N - f$ nodes do not crash, are underloaded, and all incoming threads can be delayed $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ and still be schedulable, then DQBUA meets the termination time of all threads in its eligible execution thread set, Γ , with high computable probability, P_{alg} .*

Proof. As in Lemma 4, no thread in the eligible thread set Γ will be rejected if nodes receive the broadcast START message and respond to that message on time. The probability of these two events is $bin(0, N - f, p)$ where $p = 1 - DELAY(T)$. Therefore, the probability that none of the threads in Γ are rejected is $P_{norej} = bin(0, N - f, p) \times bin(0, N - f, p)$. This means that all the sections belonging to those threads will be scheduled to meet their derived termination times. By Lemma 5, this implies that each of these threads, T_j , will meet their termination times with probability p_{suc}^j . Therefore, for a system with an eligible thread set, Γ , the probability that all threads meet their termination times if their sections meet their termination times is $P_{tmp} = \prod_{j \in \Gamma} p_{suc}^j$. The probability that all the remaining threads are executed to completion is thus $P_{alg} = P_{tmp} \times p_{norej}$. The reason for tolerating $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ delay is the same as in Theorem 6. \square

Definition 1 (Section Failure). A section, S_j^i , is said to have failed when one or more of the previous head nodes of S_j^i 's thread (other than S_j^i 's node) has crashed.

Lemma 8. *If a node hosting a section, S_j^i , of thread T_i fails (per Definition 1) at time t_f , every correct node will include handlers for thread T_i in its schedule by time $t_f + T_D + t_a$, where t_a is an implementation-specific computed execution bound for DQBUA calculated per the analysis in Theorem 3, with high, computable, probability, P_{hand}*

Proof. Since the QoS FD we use in this work detects a failed node in T_D time units [2], all nodes in the system will detect the failure of the node at time $t_f + T_D$. As a result, the DQBUA algorithm will be triggered and will exclude T_i from the system because node j will not send its schedule (lines 14-15 Algorithm 5). Consequently, Algorithm 5 will include the section handlers for this thread in the schedule. Execution of DQBUA completes in time t_a and thus all handlers will be included in the schedule by time $t_f + T_D + t_a$.

Of all these timing terms, only t_a is stochastic. From Theorem 3, we know that t_a will be obeyed with probability P_{hand} , therefore, the time bound derived above is also obeyed with probability P_{hand} . \square

Lemma 9. *If a section S_i , where $i \neq k$, fails (per Definition 1) at time t_f and section S_{i+1} is correct, then under DQBUA, its handler S_i^h will be released no earlier than S_{i+1}^h 's completion and no later than $S_{i+1}^h.tt + T + S_i^h.X - S_i^h.ex$.*

Proof. For $i \neq k$, a section's exception handler can be released due to one of two events; 1) its start time expires; or 2) an explicit invocation is made by the handler's successor.

In the first case, we know from the analysis in Section 4 that the start time of S_i^h is $S_{i+1}^h.tt + S_j^h.X + T - S_j^h.ex$. Thus, by definition, it satisfies the upper bound in the theorem. Also, since $S_j^h.X \geq S_j^h.ex$ (otherwise the handler would not be schedulable), $S_{i+1}^h.tt + S_j^h.X + T - S_j^h.ex > S_{i+1}^h.tt$, and this satisfies the lower bound of the theorem.

In the second case, an explicit message has arrived indicating the completion of S_{i+1}^h . Since the message was sent, this indicates that $S_{i+1}^h.tt$ has already passed, thus satisfying the lower bound of the theorem. In addition, the message should have arrived T time units after S_{i+1}^h finishes execution (i.e. at $S_{i+1}^h.tt + T$), since $S_{i+1}^h.tt + T \leq S_{i+1}^h.tt + T + S_i^h.X - S_i^h.ex$ (remember that $S_i^h.X \geq S_i^h.ex$), then the upper bound is satisfied. \square

An interesting thing about the property above is that it is not probabilistic in nature. At first sight, it would seem that the property is stochastic due to the probabilistic communication delay used in the second case mentioned in the proof. One would expect the upper bound in the property to be respected only probabilistically in the second case. However, if the upper bound is not met in the second case (i.e. the stochastic communication delay causes the notification of the completion of handler S_{i+1}^h to arrive after the upper bound in the theorem), then the first case kicks in and starts the handler before the upper bound expires anyway. Therefore this result is deterministic in nature.

Lemma 10. *If a section S_i fails (per Definition 1), then under DQBUA, its handler S_i^h will complete no later than $S_i^h.tt$ (barring S_i^h 's failure).*

Proof. If one or more of the previous head nodes of S_i 's thread has crashed, it implies that S_i 's thread was present in a system wide schedulable set previously constructed. This implies that S_i and its handler were previously determined to be feasible before $S_i.tt$ and $S_i^h.tt$ respectively (lines 5-7 of Algorithm 6).

When some previous head node of S_i 's thread fails, DQBUA will be triggered and will remove S_i from the pending queue. In addition, Algorithm 5 will include S_i^h in H and construct a feasible schedule containing S_i^h (lines 8-9 and line 10). Since the schedule is feasible and S_i^h is inserted to meet $S_i^h.tt$ (line 7, Algorithm 6), then S_i^h will complete by time $S_i^h.tt$. \square

Note that the termination times mentioned in the proofs above may be modified to reflect dependencies (lines 20-27 in Algorithm 6). We now state DQBUA's bounded clean-up property.

Theorem 11. *In the event of a failure of a thread, the thread's handlers will be executed in LIFO (last-in first-out) order. Furthermore, all (correct) handlers will complete in bounded time. For a thread with k sections, handler termination times $S_i^h.X$, which fails at time t_f , and (distributed) scheduler latency t_a , this bound is $T_i.X + \sum_i S_i^h.X + kT + T_D + t_a$, with high computable probability P_{exp} .*

Proof. The LIFO property follows from Lemma 9. Since it is guaranteed that each handler, S_i^h , cannot begin before the termination time of handler S_{i+1}^h (the lower bound in Lemma 9), then we guarantee LIFO execution of the handlers.

The fact that all correct handlers complete in bounded time is shown in Lemma 10, where each correct handler is shown to complete before its termination time.

Finally, if a thread fails at time t_f , all nodes will include handlers for this thread in their schedule by time $t_f + T_D + t_a$ (Lemma 8) with probability P_{hand} and DQBUA guarantees that all these sections will complete before their termination times (Lemma 10). Due to the LIFO nature of handler executions, the last handler to execute is the first exception handler, S_1^h . The termination time of this handler (from the equations in Section 4) is $T_i.X + \sum_i S_i^h.X + kT + T_D + t_a$ (which is basically the sum of the relative termination times of all the exception handlers, plus the termination time of the last section, which is used as an estimate for the worst case failure time of the threads per the discussion in Section 4, k communication delays T to notify handlers in LIFO order, T_D to detect the failure after it occurs and t_a for DQBUA to execute).

Since Lemma 10 guarantees that all handlers will finish before their derived termination times, the only stochastic part of the theorem is the probability that DQBUA will include the handlers of all the section in time $t_f + T_D + t_a$. From Lemma 8, we know this probability is P_{hand} , thus $P_{exp} = P_{hand}$. \square

Theorem 12. *A deadlock is resolved in at most $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units by terminating the thread that can contribute the least amount of utility to the system.*

Proof. A resource request is a distributed scheduling event. Therefore, when the resource request that causes a wait-for cycle to form occurs, it is handled in at most $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units (see Theorem 3). While computing the dependency chain in Algorithm 5, the cycle will be detected and broken by terminating the thread in the cycle with the lowest PUD. The theorem follows. \square

Theorem 13. *Resource contention is resolved in order of thread PUD.*

Proof. Threads are ordered according to their PUD in Algorithm 5. Therefore if more than one thread is waiting for a particular resource, threads with higher PUD will be considered before threads with lower PUD. \square

Theorem 14. *DQBUA limits thrashing by reducing the number of instances of DQBUA spawned by concurrent distributed scheduling event.*

Proof. This follows from the proof of Theorem 22 of [7]. \square

6 Experimental Results

We performed a series of simulation experiments on ns-2 to compare the performance of DQBUA to RTG-DS in terms of Accrued Utility Ratio (AUR) and Deadline Satisfaction Ratio (DSR). We define AUR as the ratio of the accrued utility (the sum of U_i for all completed threads) to the utility available (the sum of U_i for all available jobs) and DSR as the ratio of the number of threads that meet their termination time to the total number of threads in the system. We considered threads with three segments. Each thread starts at its origin node with its first segment. The second segment is a result of a remote

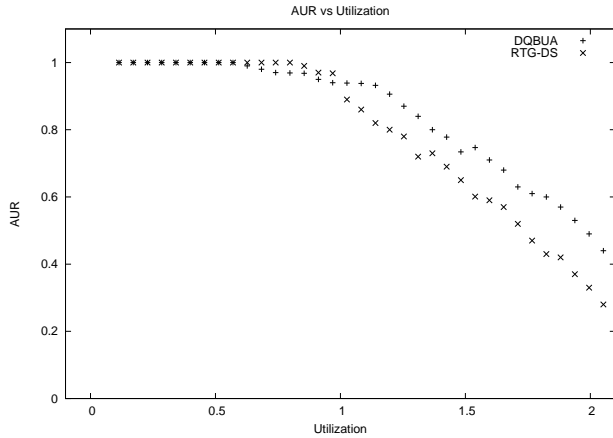


Figure 1. AUR vs. Utilization

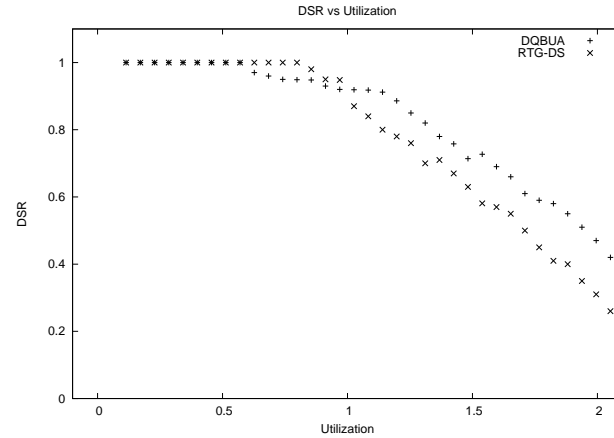


Figure 2. DSR vs. Utilization

invocation to some node in the system, and the third segment occurs when the thread returns to its origin node to complete its execution.

The periods of these threads are fixed, and we vary their execution times to obtain a range of utilization ranging from 0 to 200%. For fair comparison, all algorithms were simulated using a synchronous system model, where communication delay varied according to an exponential distribution with mean and standard deviation 0.02 seconds but could not exceed an upper bound of 0.5 seconds. Our system consisted of fifty client nodes and five servers. In our experiments, the utilization of the system is considered the *maximum* utilization experienced by any node. We assume that there are two, different, resources on each node in the system. Each section randomly choose which resource, if any, it wishes to acquire. The time spent using a resource is a uniformly distributed random number that represents a proportion of that section's remaining execution time.

DQBUA is a collaborative scheduling algorithm, as such, its strength lies in its ability to give priority to threads that will result in the most system-wide accrued utility even if the sections of those threads do not maximize local utility on the nodes they are hosted. The thread set that highlights this property contains threads that would be given low priority if local scheduling is performed but should be assigned high priority due to the system-wide utility they accrue. Therefore, we chose a thread set that contains high utility threads that have one section with above average execution time (resulting in low PUD for that section) and other sections with below average execution times (resulting in high PUD for those sections). Such thread sets test the ability of the algorithm to take advantage of collaboration to avoid making locally optimal decisions that would compromise global optimality.

As can be seen in Figures 1 and 2, the performance of DQBUA is better than that of DTG-DS during overloads. This occurs, because DQBUA performs collaborative scheduling thus maximizing, as much as possible, **system-wide** accrued utility. On the other hand, RTG-DS does not perform collaborative scheduling (but uses gossip to identify the next head node of a thread and to improve the reliability of the communication layer) and therefore performs worse during overloads.

7 Conclusion and Future Work

We presented an algorithm, DQBUA, for scheduling dependent distributable threads in a partially synchronous system. We showed that it accrues optimal utility during underloads and attempts to maximize the accrued utility during overloads. We experimentally compared DQBUA to another scheduling algorithm for dependent threads, RTG-DS, and showed that DQBUA outperforms RTG-DS during overloads. Future work include considering more dynamic networks such as mobile ad hoc networks and finding more sophisticated methods for breaking a wait-for graph when distributed deadlock is detected.

References

- [1] J. R. Cares. *Distributed Networked Operations: The Foundations of Network Centric Warfare*. iUniverse, Inc., 2006.
- [2] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, 2002.

- [3] R. Clark, E. Jensen, and F. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *1993 Winter USENIX Conf.*, pages 127–146, 1993.
- [4] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, CMU, 1990. CMU-CS-90-155.
- [5] P. H. Dana. Global positioning system (gps) time dissemination for real-time applications. *Real-Time Syst.*, 12(1):9–40, 1997.
- [6] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HOTOS '01*, pages 75–80, 2001.
- [7] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Fast scheduling of distributable real-time threads with assured end-to-end timeliness. Technical report, Virginia Tech, ECE Dept., November 2007. Available at: http://www.real-time.ece.vt.edu/RST_TR.pdf.
- [8] W. A. Halang and M. Wannemacher. High accuracy concurrent event processing in hard real-time systems. *Real-Time Syst.*, 12(1):77–94, 1997.
- [9] K. Han, B. Ravindran, and E. D. Jensen. Exploiting slack for scheduling dependent, distributable real-time threads in mobile ad hoc networks. In *RTNS 2007*, pages 225–234, 2007.
- [10] E. Jensen, C. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems, 1985. IEEE RTSS, pages 112–122, 1985.
- [11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
- [12] D. P. Maynard, S. E. Shipman, et al. An example real-time command, control, and battle management application for alpha. Technical Report Archons Project 88121, CMU CS Dept., December 1988.
- [13] B. Sterzbach. GPS-based clock synchronization in a mobile, distributed real-time system. *Real-Time Syst.*, 12(1):63–75, 1997.