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

Sherif Fahmy, Binoy Ravindran, and E. D. Jensen

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. DBQUA 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

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

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

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 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 [6] 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 [8], a dependent thread scheduling algorithm that uses gossip ro 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. Distributable threads execute in local and remote objects by location-independent invocations and returns. 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 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.

We assume that execution time estimates of sections of a thread are known when it arrives into the system. The sequence of remote invocations and returns made by a thread can typically be estimated by analyzing the thread code. 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 $\mathbf{T} = \{T_1, T_2, \ldots\}$ and the set of sections of a thread T_i is denoted as $[S_1^i, S_2^i, \ldots, S_k^i]$. See [7] for more details.

Timeliness Model. A thread's time constraint is expressed using a Time/Utility Function (TUF) [9]. A TUF decouples the urgency of a thread from its importance. This is useful 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 $\Pi^c = \{1, 2, \cdots, N\}$ and a set of server nodes $\Pi = \{1, 2, \cdots, n\}$ (*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 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 thread sections on the node and a scheduling co-processor as in [3]), have access to GPS clocks that provides each node with a UTC time-source with nanosecond accuracy (e.g., [11]) and are equipped with appropriately tuned QoS failure detectors (FDs) [2] (see [7] for further details).

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 either of these events occur, exception handlers are triggered to restore the system to a safe state. The exception handlers we consider have time constraints expressed as relative deadlines. See [7] for details.

Failure Model. 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, [5] — 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 $f^s_{max} \leq n/3$ server failures (see [6]). The actual number of failures is denoted as $f^s \leq f^s_{max}$ for servers and $f \leq f_{max}$ where $f_{max} \leq N-1$ for clients.

Resource Model. Threads can access serially reusable non-CPU resources located at their nodes during their execution. We consider the single resource model — i.e., a thread cannot have more than one outstanding request at any given instance of time. Resources are shared under mutual exclusion constrains and a thread explicitly releases all granted resources before termination. Threads are assumed to access their resources in 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.

Resource request/release pairs are assumed to be confined within one node, 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. Such a lock is released when the thread's head returns back to the node on which the resource was acquired.

Scheduling Objectives. Our primary objective is to design a thread scheduling algorithm to 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 a deadlock.

3 Algorithm Rationale

In [6], we develop QBUA, a scheduling algorithm for real-time threads in partially synchronous systems. Here, we extend QBUA to handle resource dependencies and precedence constraints, we call the resulting algorithm DQBUA. As in [4], precedence constraints can be programmed as resource dependencies and are handled the same way. When a node detects a distributed scheduling event (the failure of a node, the arrival of a new thread or a resource request) it contacts a quorum system requesting permission to run an instance of DQBUA. Once permission is granted, it broadcasts a message to all other nodes requesting their scheduling information. When the requesting node receives this information, 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 [6] 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].

DBQUA handles both distributed and local deadlock using a deadlock detection and resolution protocol that ensures that deadlocks are resolved in a timely manner and that the loss in accrued system utility is minimized when deadlocks are resolved.

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 sends a message to all other nodes requesting their scheduling information. The node then waits for $2T$ time units to receive replies and then invokes Algorithm 3 to construct a system wide schedule using the collected information. Algorithm 3 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 schedule. 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 handlers if necessary).

We define the global PUD of a thread as the ratio of the utility of the thread to the total remaining executing time of its sections (see [7] for details). Therefore, global PUD is a measure of the "return on investment" of that thread, [4] shows that considering threads in non-decreasing order of PUD maximizes accrued utility.

In the absence of dependencies, the global PUD of a thread represents the utility that would be accrued if a thread where to execute immediately. However, in the presence of dependencies, the utility of a thread can only be accrued if all threads it depends on are scheduled first. Thus, when a section requests a resource, we compute its dependency chain 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 chain 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 chain 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]. 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 1: computePUD

```
1:  Input: $T_i$, $Dep(i,k)$, $j$; // $j$: where request occured
2:  $Ut \leftarrow 0$; $Time \leftarrow 0$; $Seen \leftarrow \emptyset$;
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 $S$ till last visit to $j$
8:              $S.Rem \leftarrow \Sigma_{k \in \Gamma_1} RE_k^{S.ID}$;
9:              //$\Gamma_2$: all downstream sections
10:             $S.Abort \leftarrow \Sigma_{k \in \Gamma_2} S_k^h.ex$;
11:             if $S.Abort > S.Rem$ then
12:                 $Time \leftarrow Time + S.Rem$;
13:                 $Ut \leftarrow Ut + U_T(t_{curr} + S.Rem)$
14:             else $Time \leftarrow Time + S.Abort$;
15: $Time \leftarrow Time + GE_i$; $Ut \leftarrow Ut + U_i(t_{curr} + GE_i)$;
16: $T_i.PUD = Util/Time$; return $T_i.PUD$;
```

Algorithm 2: isFeasible

```
1:  Input: $\sigma_i$; //Schedule for each node
2:  for $1 \le i \le N$ do
3:      $pos_i \leftarrow 1$;
4:  Until $(pos_i = \text{length}(\sigma_i)\ ,\ 1 \le i \le N)$ do
5:      for $1 \le i \le N$ 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:                 return $false$;
15:             $pos_i \leftarrow pos_i + 1$;
16:     return $true$;
```

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 the termination time of the entire thread. The termination time of other sections is the latest start time of the section's successor minus the communication delay.

Similarly, we drive the termination times of exception handlers as the sum of their start time and their execution time. However, we perform the decomposition backwards starting with the termination time of the last handler which is computed as the termination time of that handler's section plus the execution time of the handler. The termination time of other handlers is the latest termination time of the handler's successor plus the communication delay plus the handler's execution time. This ensures that handler termination times are arranged in LIFO order. See [7] for more details. Using these derived termination times, we can check a schedule's feasibility.

Algorithm 3: ConstructSchedule

```
1: input: $\Gamma$; //Set of threads in the system
2: input: $\sigma_j^p$, $H_j \leftarrow$ 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:         Compute $Dep(i, j)$, resolving deadlock if necessary;
7:         $T_i.PUD \leftarrow$ ComputePUD($T_i$, $Dep(i, j)$);
8: for each task $el \in \sigma_j^p$ do
9:     if el is an exception handler for section $S_j^i$ then Insert($el$, $H_j$, $el.tt$);
10: $\sigma_j \leftarrow H_j$;
11: $\sigma_{temp} \leftarrow$ 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 $T_i.stop \leftarrow true$;
15:     if $T_i.PUD > 0$ and $T_i.stop \neq true$ then insertByEDF($T_i$, $Dep(i, j)$);
16: for each $j \in N$ do
17:     if $\sigma_j \neq \sigma_j^p$ then Mark node $j$ as being affected;
```

Algorithm 2 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 in the system when Algorithm 2 is invoked. Therefore we need to estimate the start time of these sections when computing their estimated completion time.

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 2 is run.

We use this relatively expensive method for checking the feasibility of schedules since alternative methods can be misleading. The expedient method, used in some previous work, of using a section's latest start time (computed as its predecessor's latest termination time plus a communication delay, $S_{i-1}.tt + T$) as an estimate for its start time means that the section will have no slack. Thus, the section cannot tolerate any interference by other sections. This leads to pessimistic results with some threads being rejected from an underloaded system. Algorithm 2 handles this by computing a better estimate of section start times, albeit at a higher cost.

In Algorithm 3, each node, j, sends the node running DQBUA its current local schedule σ_j^p. Using these schedules, the set of threads in the system, Γ, is derived. In lines 3-8, DQBUA computes the global PUD of each thread in Γ. If a section belonging to a thread cannot meet its termination time if it were scheduled immediately, the thread is assigned a PUD of zero since it cannot possibly accrue any utility to the system (line 4). Otherwise, we compute the dependency chain for the thread's sections and call Algorithm 1 to compute its global PUD (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.

Algorithm 4: 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($S_j^i$,$\sigma_j^{tmp}$,$S_j^i.tt$); $TT_{cur} \leftarrow S_j^i.tt$;
6:          if $S_j^h \notin \sigma_j^p$ then Insert($S_j^h$,$\sigma_j^{tmp}$,$S_j^h.tt$);
7:          for $\forall S_n^k \in Dep(i,j)$ do
8:              if $S_n^k \in \sigma_n^{tmp}$ then
9:                  if $S_n^k$ is an abortion handler then Remove all sections belonging to $S_n^k$'s thread;
10:                 $TT \leftarrow$ lookUp($S_n^k$, $\sigma_n^{tmp}$);
11:                 if $TT < TT_{cur}$ then $TT_{cur} \leftarrow TT$; Continue;
12:                 else
13:                     Remove($S_n^k$, $\sigma_n^{tmp}$, $TT$); Insert($S_n^k$, $\sigma_n^{tmp}$, $TT_{cur}$); $\delta \leftarrow TT - TT_{cur}$;
14:                     for all predecessors, $S_l^x$, of $S_n^k$ do
15:                         //If $S_n^k$ is an abortion handler, $S_l^x$s are also abortion handlers.
16:                         //Otherwise, $S_l^x$s are normal sections
17:                         $TT \leftarrow$ lookUp($S_l^x$, $\sigma_l^{tmp}$); $\gamma \leftarrow \delta$;
18:                         if $S_n^k.tt - TT < \delta$ then $\gamma \leftarrow \delta - (S_n^k.tt - TT)$ ;
19:                         Remove($S_l^x$, $\sigma_l^{tmp}$, $TT$); Insert($S_l^x$, $\sigma_l^{tmp}$, $TT - \gamma$);
20:             else
21:                 $TT_{cur} \leftarrow \min(TT_{cur}, S_n^k.tt)$; Insert($S_n^k$, $\sigma_n^{tmp}$, $TT_{cur}$);
22:                 if $S_n^k$ is not an abortion handler and $S_n^h \notin \sigma_n^p$ then Insert($S_n^h$,$\sigma_n^{tmp}$,$S_n^h.tt$);
23: if isFeasible($\sigma_j^{tmp}$'s)=true then $\sigma_j \leftarrow \sigma_j^{tmp}$ for all $j$ ;
24: return $\sigma_j$ for all $j$;
```

It is necessary to ensure that the exception handlers of any thread that has been accepted into the system can meet their termination time to ensure that the system is restored to a safe state if the thread fails. This is done 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 maintains the feasibility of a schedule as an algorithm invariant, these handlers will meet their termination times. In line 11, we sort the threads in non-increasing order of PUD and consider them for scheduling in that order (lines 12-15). In line 14 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 has not been rejected from the system, then we insert its sections, and their dependencies, into the scheduling queue of the nodes responsible for them in non-decreasing order of termination time by calling Algorithm 4 (lines 15).

When Algorithm 4 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). For each of the sections of the current thread, 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 tentatively inserted into the schedule (lines 5-6).

We then consider the dependencies of that section (lines 7-20). 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 a deadline ordered list. So during underloads, DQBUA defaults to Earliest Deadline First (EDF) scheduling, which is an optimal realtime scheduling algorithm [10] 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 (line 9).

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 (line 13). In addition, all its predecessors have their termination time adjusted to reflect this new value (lines 14-19). Finally, the feasibility of the tentative schedule is checked (line 23) and the changes are made permanent if the schedule is feasible.

5 Algorithm Properties

We establish several properties of DQBUA. Due to space limitations, some of the properties and all of the proofs are omitted here, and can be found in [7]. 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.

Theorem 1 *A distributed scheduling event is handled at most $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units after it occurs, with high, computable, probability, P_{hand}.*

Theorem 2 *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}.*

Theorem 3 *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}.*

Theorem 4 *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.*

Theorem 5 *Resource contention is resolved in order of thread PUD.*

Theorem 6 *DQBUA limits thrashing by reducing the number of instances of DQBUA spawned by concurrent distributed scheduling events.*

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

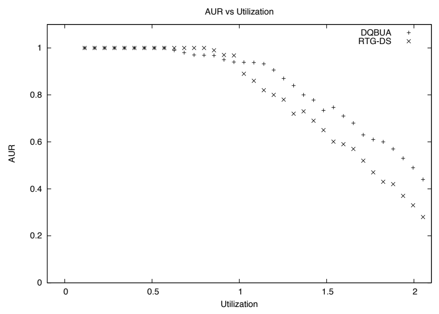

Fig. 1 AUR vs. Utilization

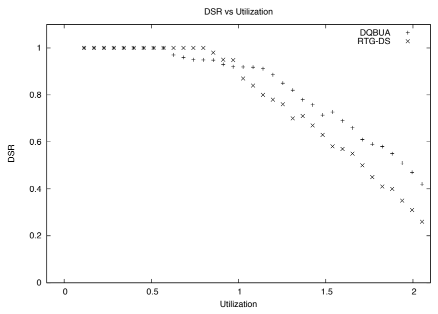

Fig. 2 DSR vs. Utilization

The periods of the threads are fixed, and we vary their execution times to obtain a range of utilization 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 and an upper bound of 0.5 seconds. Our system consisted of fifty client nodes and five servers. System utilization is considered the *maximum* utilization experienced by any node. We assume that there are two, different, resources on each node. A section randomly chooses which resource, if any, it wishes to acquire and the time spent holding a resource is a uniformly distributed random number that represents a proportion of that section's remaining execution time. See [7] for more details.

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. RTG-DS does not perform collaborative scheduling and therefore performs worse during overloads. However, during underloads, RTG-DS outperforms DQBUA as the utilization of the system approaches one, since DQBUA has higher overhead.

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. However, during underloads, RTG-DS has better performance since it has lower overhead. Future work includes 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. Cares, J.R.: Distributed Networked Operations: The Foundations of Network Centric Warfare. iUniverse, Inc. (2006)
2. Chen, W., Toueg, S., Aguilera, M.K.: On the quality of service of failure detectors. IEEE Transactions on Computers **51**(1), 13–32 (2002)
3. Clark, R., Jensen, E., Reynolds, F.: An architectural overview of the alpha real-time distributed kernel. In: 1993 Winter USENIX Conf., pp. 127–146 (1993)
4. Clark, R.K.: Scheduling dependent real-time activities. Ph.D. thesis, CMU (1990). CMU-CS-90-155
5. Druschel, P., Rowstron, A.: PAST: A large-scale, persistent peer-to-peer storage utility. In: HOTOS '01, pp. 75–80 (2001)
6. Fahmy, S.F., Ravindran, B., Jensen, E.D.: Fast scheduling of distributable real-time threads with assured end-to-end timeliness. Tech. rep., Virginia Tech, ECE Dept. (2007). Available at: `http://www.real-time.ece.vt.edu/RST_TR.pdf`
7. Fahmy, S.F., Ravindran, B., Jensen, E.D.: Scheduling dependent distributable real-time threads in dynamic networked embedded systems (2007). Available at: `http://filebox.vt.edu/users/fahmy/TR-DIPES.pdf`
8. Han, K., Ravindran, B., Jensen, E.D.: Exploiting slack for scheduling dependent, distributable real-time threads in mobile ad hoc networks. In: RTNS 2007, pp. 225–234 (2007)
9. Jensen, E., Locke, C., Tokuda, H.: A time driven scheduling model for real-time operating systems (1985). IEEE RTSS, pages 112–122, 1985.
10. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM **20**(1), 46–61 (1973)
11. Sterzbach, B.: GPS-based clock synchronization in a mobile, distributed real-time system. Real-Time Syst. **12**(1), 63–75 (1997)