Fast Scheduling of Distributable Real-Time Threads with Assured End-to-End Timeliness

Sherif F. Fahmy[1], Binoy Ravindran[1], and E. D. Jensen[2]

[1] ECE Dept., Virginia Tech, Blacksburg, VA 24061, USA
[2] The MITRE Corporation, Bedford, MA 01730, USA

Abstract. We consider networked, embedded real-time systems that operate under run-time uncertainties on activity execution times and arrivals, node failures, and message losses. We consider the distributable threads abstraction for programming and scheduling such systems, and present a thread scheduling algorithm called QBUA. We show that QBUA satisfies (end-to-end) thread time constraints in the presence of crash failures and message losses, has efficient message and time complexities, and lower overhead and superior timeliness properties than past thread scheduling algorithms. Our experimental studies validate our theoretical results, and illustrate the algorithm's effectiveness.

1 Introduction

Some emerging, networked embedded real-time systems (e.g., US DoD's Network Centric Warfare systems [1]) are subject to resource overloads (due to context-dependent activity execution times), arbitrary activity arrivals, and arbitrary node failures and message losses. Reasoning about *end-to-end* timeliness is a difficult and unsolved problem in such systems. A distinguishing feature of such systems is their relatively long activity execution time scales (e.g., milliseconds to minutes), which permits more time-costlier 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 optimize system-wide end-to-end timeliness. The *distributable thread* programming abstraction which first appeared in the Alpha OS [2], and later 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/scheduling abstraction, and hereafter, refer to them as *threads*, except as necessary for clarity.

Contributions. In this paper, we consider the problem of scheduling threads in the presence of the previously mentioned uncertainties, focusing particularly on (arbitrary) node failures and message losses. Past efforts on thread scheduling

(e.g., see [3] 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. In this work, we compare QBUA to three previous thread scheduling algorithms, HUA, CUA, and ACUA (see [3] and references therein).

HUA is an independent scheduling algorithm, which sometimes may make locally optimal decisions that may not be globally optimal. This is overcome by CUA and ACUA, which are collaborative scheduling algorithms that use uniform consensus [4] for unanimously deciding on system-wide thread schedules in the presence of node failures. In [3], it is shown that ACUA has superior timeliness properties (e.g., lower number of missed deadlines) than HUA and CUA. In addition, HUA and CUA consider synchronous computational models (i.e., those with deterministically bounded time variables). In contrast, ACUA considers the partially synchronous model in [5], where message delay and message loss are probabilistically described. Though this increases ACUA's coverage[3], the algorithm has high overhead, thereby only allowing threads that can tolerant this large overhead to reap the algorithm's superior timeliness.

In this paper, we present a collaborative scheduling algorithm called the *Quorum-Based Utility Accrual scheduling* (or QBUA) that precisely overcomes ACUA's overhead disadvantage. The algorithm considers the partially synchronous model in [5], and uses a Quorum set of nodes for majority agreement on constructing system-wide thread schedules. We show that QBUA satisfies thread time constraints in the presence of node crash failures and message losses, has efficient message and time complexities that compare favorably with other algorithms in its class, and lower overhead and superior timeliness than past algorithms including CUA and HUA. We also show that the algorithm's lower overhead enables it to allow more threads to benefit from its superior timeliness, than that allowed by past algorithms.

2 Models and Objective

Distributable Thread Abstraction. Distributable threads, our computing abstraction, 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

[3] As defined in [6], coverage is the decreasing likelihood for the algorithm's timing assurances to be violated, when the underlying synchrony assumptions are violated at run-time (e.g., due to overloads or other exigencies). This likelihood reduces when coverage increases.

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. We assume that execution time estimates of sections of a thread are known when the thread arrives into the system and are described using TUFs (see our timeliness model). 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. We specify the time constraint of each thread using a Time/Utility Function (TUF) [8]. 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. Our system consists of a set of client nodes $\coprod = \{1, 2, \cdots, N\}$ and a set of server nodes $\mathit{\Pi} = \{1, 2, \cdots, n\}$ (*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 the basic communication channels may loose 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 [2]. We also assume that nodes in our system have access to GPS clocks that provides each node with a UTC time-source with high precision (e.g., [9]) and are equipped with appropriately tuned QoS FDs [5]. Further details about our system model are provided in [7].

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. In the case of time constraint-violation or node failure, these 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 more details.

Failure Model. The nodes are subject to crash failures. When a process crashes, it loses it 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, [10] — 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 never fails; it is *faulty* if it is not correct. QBUA tolerates up to $N-1$ client failures and up to $f^s_{max} \leq n/3$ server failures. The actual number of server failures is denoted as $f^s \leq f^s_{max}$ and the actual number of client failures is denoted as $f \leq f_{max}$ where $f_{max} \leq N-1$.

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. The algorithm must also provide assurances on the satisfaction of thread termination times in the presence of (up to f_{max}) crash failures. Moreover, the algorithm must exhibit the best-effort property (see Section 1 of [7] for details).

3 Algorithm Rationale

QBUA is a collaborative scheduling algorithm, which allows it to construct schedules that result in higher system-wide accrued utility by preventing locally optimal decisions from compromising system-wide optimality. It also allows QBUA to respond to node failures by eliminating threads that are affected by the failures, thus allowing the algorithm to gracefully degrade timeliness in the presence of failures. There are two types of scheduling events that are handled by QBUA, viz: a) local scheduling events and b) distributed scheduling events.

Local scheduling events are handled locally on a node without consulting other nodes. Examples of local scheduling events are section completion, section handler expiry events etc. For a full list of local scheduling events please see Algorithm 7 in [7]. Distributed scheduling events need the participation of all nodes in the system to handle them. In this work, only two distributed scheduling events exit, viz: a) the arrival of a new thread into the system and b) the failure of a node. A node that detects a distributed scheduling event sends a START message to all other nodes requesting their scheduling information so that it can compute a System Wide Executable Thread Set (or SWETS). Nodes that receive this message, send their scheduling information to the requesting node and wait for schedule updates (which are sent to them when the requesting node computes a new system-wide schedule). This may lead to contention if several different nodes detect the same distributed scheduling event concurrently.

For example, when a node fails, many nodes may detect the failure concurrently. It is superfluous for all these nodes to start an instance of QBUA. In addition, events that occur in quick succession may trigger several instances of QBUA when only one instance can handle all of those events. To prevent this, we use a quorum system to arbitrate among the nodes wishing to run QBUA. In order to perform this arbitration, the quorum system examines the time-stamp of incoming events. If an instance of QBUA was granted permission to run *later*

than an incoming event, there is no need to run another instance of QBUA since information about the incoming event will be available to the version of QBUA already running (i.e., the event will be handled by that instance of QBUA).

4 Algorithm Description

As mentioned above, whenever a distributed scheduling event occurs, a node attempts to acquire permission from the quorum system to run a version of QBUA. After the quorum system has arbitrated among the nodes contending to execute QBUA, the node that acquires the "lock" executes Algorithm 1. In Algorithm 1, the node first broadcasts a start of algorithm message (line 1) and then waits $2T$ time units[4] 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 4. After computing SWETS, the node contacts affected nodes (i.e. nodes that will have sections added or removed from their schedule as a result of the scheduling event).

Algorithm 1: 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 2 shows the details of the algorithm that client nodes run when attempting to acquire a "lock" on running a version of QBUA. The algorithm is loosely based on Chen's solution for FTME [11]. Upon the arrival of a distributed scheduling event, a node tries to acquire a "lock" on running QBUA (the try_1 part of the algorithm that starts on line 3).

The first thing that the node does (lines 4-5) is check if it is currently running an instance of QBUA that is in its information collection phase (line 2 in Algorithm 1). If so, the new event that has occurred can simply be added to the information being collected by this version of QBUA. However, if no current instance of QBUA is being hosted by the node, or if the instance of QBUA being hosted has passed its information collection phase, then the event may have to spawn a new instance of QBUA (this starts at line 6 in the algorithm).

The first thing that Algorithm 2 does in this case is send a time-stamped request to the set of server nodes, Π, in the system (lines 8-10). The time-stamp is used to inform the quorum nodes of the time at which the event was detected by the current node. Beginning at line 3, Algorithm 2 collects replies from the servers. Once a sufficient number of replies have arrived (line 14), Algorithm 2 checks whether its request has been accepted by a sufficient ($\lceil \frac{2n}{3} \rceil$ see Section 5) number of server nodes. If so, the node computes SWETS (lines 15-16).

On the other hand, if an insufficient number of server nodes support the request, two possibilities exist. The first possibility is that another node has

[4] T is communication delay derived from the random variable describing the communication delay in the system.

been granted permission to run an instance of QBUA to handle this event. In this case, the current node does not need to perform any additional action and so releases the "lock" it has acquired on some servers (lines 17-21).

The second possibility is that the result of the contention to run QBUA at the servers was inconclusive due to differences in communication delay. For exa

Algorithm 2: QBUA on client node i

```
1:  timestamp; // time stamp variable initially set to nil
2:  upon thread arrival or detection of a node failure:
3:    try_1:
4:      if a current version of QBUA is waiting for information from other nodes then
5:          Include information about event when computing SWETS;
6:      else
7:          timestamp ← GetTimeStamp;
8:          for all r_j ∈ Π do
9:              resp[j] ← (nil, nil);
10:             send (REQUEST, timestamp) to r_j;
11:         repeat
12:             wait until [received (RESPONSE, owner, t) from some r_j];
13:             if (c_1 ≠ owner or timestamp = t) then resp[j] ← (owner, t);
14:             if among resp[], at least m of them are not (nil, nil) then
15:                 if at least m elements in resp[] are (c_1, t) then
16:                     return Compute SWETS;
17:                 else if at least m elements in resp[] agree about a certain node then
18:                     for all r_k ∈ Π such that resp[k] ≠ (nil, nil) do
19:                         if resp[k].owner = c_1 then
20:                             send (RELEASE,timestamp) to r_k;
21:                     Skip rest of algorithm; //Event is already being handled
22:                 else
23:                     for all r_k ∈ Π such that resp[k] ≠ (nil, nil) do
24:                         if resp[k].owner = c_1 then
25:                             send (YIELD,timestamp) to r_k;
26:                         else
27:                             send (INQUIRE,timestamp) to r_k;
28:                         resp[k] ← (nil, nil);
29:         until forever ;
30:       exit_1:
31:         oldtimestamp ← timestamp;
32:         timestamp ← GetTimeStamp;
33:         for all r_k ∈ Π do
34:             send (RELEASE, oldtimestamp) to r_j;
35:             return;
36: upon receive (CHECK, t) from r_j
37:   if for all instances of QBUA running on this node, timestamp ≠ t then
38:       send (RELEASE, t) to r_j;
39: upon receive (START) from some client node
40:   Update RE_j^i for all sections;
41:   send σ_j and RE_j^i's to requesting node;
```

mple, assume that we have 5 servers and three clients wishing to run QBUA and all three clients send their request to the servers at the same time, also assume different communication delay between each server and client. Due to these communication differences, the messages of the clients may arrive in such a pattern so that two servers support client 1, another 2 servers support client 2 and the last server supports client 3. This means that no client's request is supported by a sufficient — i.e., $\frac{2n}{3}$ — number of server nodes. In this case, the

client node sends a YIELD message to servers that support it and an INQUIRE message to nodes that do not support it (line 22-28) and waits for more responses from the server nodes to resolve this conflict. Lines 30-35 release the "lock" on servers after the client node has computed SWETS, lines 36-38 are used to handle the periodic cleanup messages sent by the servers and lines 39-41 respond to the START of algorithm message (line 1, Algorithm 1).

Algorithm 3: QBUA on server node i

```
1:  c_owner[]; Array of nodes holding lock to run QBUA
2:  t_owner[]; t_owner[i] contains time-stamp of event that triggered QBUA for node in c_owner[i]
3:  t_grant[]; t_grant[i] contains time at which node in c_owner[i] was granted lock to run QBUA
4:  R_wait[]; R_wait[i] is waiting queue for instance of QBUA being run by c_owner[i];
5:  upon receive (tag, t)
6:    CurrentTime ← GetTimeStamp;
7:    if (c_1, t') appears in (c_owner[],t_owner[]) or R_wait[] then
8:      if t < t' then Skip rest of algo; //This is an old message
9:    if tag = REQUEST then
10:     if ∃ t_grant ∈ t_grant[] such that t ≤ t_grant then
11:       send (RESPONSE, c, t_grant) to c_1; //where c ← c_owner[i], such that
            t_grant[i] = t_grant;
12:       Enqueue (c_1, t) in R_wait[i], such that t_grant[i] = t_grant;
13:       Skip rest of algorithm;
14:     else
15:       AddElement(c_owner[], c_1);
16:       AddElement(t_owner[], t);
17:       AddElement(t_grant[], CurrentTime);
18:       send (RESPONSE, c_1, t) to c_1;
19:   else if tag = RELEASE then
20:     Delete entry corresponding to c_1, t from c_owner[], t_owner[], t_grant[], and R_wait[];
21:   else if tag = YIELD then
22:     if (c_1, t) ∈ (c_owner[], t_owner[]) then
23:       For i, such that (c_1, t) = (c_owner[i], t_owner[i])
24:       Enqueue (c_1, t) in R_wait[i];
25:       (c_wait, t_wait) ← top of R_wait[i];
26:       c_owner[i] ← c_wait; t_owner[i] ← t_wait;
27:       t_grant[i] ← CurrentTime;
28:       send (RESPONSE, c_wait, t_wait) to c_wait;
29:     if c_1 ∉ c_owner[] then
30:       (c, t_p) ← (c_owner[i], t_owner[i]), for min i such that t ≤ t_grant[i];
31:       send (RESPONSE, c, t_p) to c_1;
32:   else if tag = INQUIRE then
33:     (c, t_p) ← (c_owner[i], t_owner[i]), for min i such that t ≤ t_grant[i];
34:     send (RESPONSE, c, t_p) to c_1;
35: upon suspect that c_owner[i] has failed:
36:   HandleFailure(c_owner[i],c_owner[], t_owner[],t_grant[],R_wait[]);
37: periodically:
38:   ∀ c_owner ∈ c_owner[]:
39:     send (CHECK, t_owner) to c_owner; //NB. t_owner is the entry in t_owner[] that
          corresponds to c_owner.
```

Algorithm 3 is run by the servers, the function of this algorithm is to arbitrate among the nodes contending to run QBUA so as to minimize the number of concurrent executions of the algorithm. Since there may be more than one instance of QBUA running at any given time, the server nodes keep track of these instances using three arrays. The first array, $c_{owner}[]$, keeps track of which nodes are running instances of QBUA, the second, $t_{owner}[]$, stores the time at which a node in $c_{owner}[]$ sends a request to the servers (i.e., the time at which that node detects a certain scheduling event), and $t_{grant}[]$ keeps track of the time at which

server nodes grant permission to client nodes to execute QBUA. In addition, a waiting queue for each running instance of QBUA is kept in $R_{wait}[]$.

When a server receives a message from a client node, it first checks to see if this is a stale message (which may happen due to out of order delivery). A message from a client node, c_1, that has a time-stamp older than the last message received from c_1 has been delivered out of order and is ignored (line 7-8). Starting at line 9, the algorithm begins to examine the message it has received. If it is a REQUEST message, the server checks if the time-stamp of the event triggering the message is *less* than the time at which a client node was *granted* permission to run an instance of QBUA. If such an instance exists, a new instance of QBUA is not needed since the event will be handled by that previous instance of QBUA. Algorithm 3, inserts the incoming request into a waiting queue associated with that instance of QBUA and sends a message to the client (lines 10-13).

However, if no current instance of QBUA can handle the event, a client's request to start an instance of QBUA is granted (lines 14-18). If a client node sends a YIELD message, the server revokes the grant it issued to that client and selects another client from the waiting queue for that event (lines 21-31). This part of the algorithm can only be triggered if the result of the first round of contention to run QBUA is inconclusive (as discussed when describing Algorithm 2). Recall that this inconclusive contention is caused by different communication delays that allow different requests to arrive at different severs in different orders. However, all client requests for a particular instance of QBUA are queued in $R_{wait}[]$, therefore, when a client sends a YIELD message, servers are able to choose the highest priority request (which we define as the request with the earliest time-stamp and use node id as a tie breaker). Thus, we guarantee that this contention will be resolved in the second round of the algorithm. Lines 32-34 show servers' response to INQUIRE messages and lines 35-39 show the clean up procedures to remove stale messages. See [7] for how we handle failures (line 36).

Algorithm 4 is used by a client node to compute SWETS once it has received information from all other nodes in the system (line 2 in Algorithm 1). It performs two basic functions, first, it computes a system wide order on threads by computing their global Potential Utility Density (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 $\{S_1^i, S_2^i, \cdots, S_k^i\}$. 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 $\{RE_1^i, RE_2^i, \cdots, RE_k^i\}$ 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". This allows us to consider the threads for scheduling in an order that is designed to maximize accrued utility [12].

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 (see Lemma 6 in [7]).

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). In order 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 [5]. From this termination time decomposition, 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. In Algorithm 4, each node, j, sends the node running QBUA 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 4). In lines 3-6, the algorithm computes the global PUD of each thread in Γ.

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 7-9). Since these handlers were part of σ_j^p, and QBUA always maintains the feasibility of a schedule as an algorithm invariant, we are sure that these handlers will execute to completion before their termination times.

In line 10, we sort the threads in the system in non-increasing order of PUD and consider them for scheduling in that order (lines 11-21). In lines 13-14 we mark as failed any thread that has a section hosted on a node that does not participate in the algorithm. If the thread can contribute non-zero utility to the system and the thread has not been rejected from the system, then we insert its sections into the scheduling queue of the node responsible for them (line 17).

Algorithm 4: ConstructSchedule

```
1:  input: Γ; //Set of threads in the system
2:  input: σ_j^p, H_j ← nil; //σ_j^p: Previous schedule of node j, H_j: set of handlers scheduled
3:  for each T_i ∈ Γ do
4:      if for some section S_j^i belonging to T_i, t_curr + S_j^i.ex > S_j^i.tt then
5:          T_i.PUD ← 0;
6:      else T_i.PUD ← U_i(t_curr+GE_i) / GE_i;
7:  for each task el ∈ σ_j^p do
8:      if el is an exception handler for section S_j^i then Insert(el, H_j, el.tt);
9:  σ_j ← H_j;
10: σ_temp ← sortByPUD(Γ);
11: for each T_i ∈ σ_temp do
12:     T_i.stop ←false;
13:     if did not receive σ_j from node hosting one of T_i's sections S_j^i then
14:         T_i.stop ←true;
15:     for each remaining section, S_j^i, belonging to T_i do
16:         if T_i.PUD > 0 and T_i.stop ≠true then
17:             Insert(S_j^i, σ_j, S_j^i.tt);
18:             if S_j^h ∉ σ_j^p then Insert(S_j^h, σ_j, S_j^h.tt);
19:             if isFeasible(σ_j)=false then
20:                 T_i.stop ←true;
21:                 Remove(S_k^i, σ_k, S_k^i.tt) for 1 ≤ k ≤ j;
22:                 if S_j^i ∉ σ_j^p then Remove(S_j^h, σ_j, S_j^h.tt);
23: for each j ∈ N do
24:     if σ_j ≠ σ_j^p then Mark node j as being affected by current scheduling event;
```

After inserting the section into its corresponding ready queue (at a position reflecting its termination time), we check to see whether this section's handler had been included in the previous schedule of the node. If so, we do not insert the handler into the schedule since this has been already taken care of by lines 7-8. Otherwise, the handler is inserted into its corresponding ready queue (line 18). Once the section, and its handler, have been inserted into the ready queue, we check the feasibility of the schedule (line 19). If the schedule is infeasible, we remove the thread's sections from the schedule (line 21). However, we first check to see whether the section's handler was part of a previous schedule before we remove it (line 22). We perform this check before removing the handler because if the handler was part of a previous schedule, then its section has failed and we should keep its exception handler for clean up purposes. Finally, if the schedule

of any node has changed, these nodes are marked to have been affected by the current instance of QBUA (lines 23-24). It is to these nodes that the current node needs to multicast the changes that have occurred (line 4, Algorithm 1). In order to test the feasibility of a schedule, we need to check if all the sections in the schedule can complete before their derived termination times.

The full algorithm is depicted in Algorithm 6 in [7]. QBUA's dispatcher is shown in Algorithm 7 in [7]. Only two scheduling events result in collaborative scheduling, viz: the arrival of a thread into the system, and the failure of a node, all other scheduling events are handled locally. Since we are talking about a partially synchronous system, the FD we use to detect node failures can make mistakes. Thus, QBUA may be started due to an erroneous detection of failure. The this can be reduced by designing a QoS FD [5] with appropriate QoS parameters.

5 Algorithm Properties

We establish several properties of QBUA. 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, and Γ is the set of threads in the system.

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

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

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

Lemma 4. *The worst case message complexity of the algorithm is* $O(n + N)$.

Theorem 5. *If all nodes are underloaded, no nodes fail (i.e.* $f = 0$*) and each thread can be delayed* $O(T+N+|\Gamma| \log(|\Gamma|))$ *time units once and still be schedulable, QBUA meets all the thread termination times yielding optimal total utility with high, computable, probability,* P_{alg}.

Theorem 6. *If* $N - f$ *nodes do not crash, are underloaded, and all incoming threads can be delayed* $O(T + N + |\Gamma| \log(|\Gamma|))$ *and still be schedulable, then QBUA meets the execution time of all threads in its eligible execution thread set,* Γ*, with high computable probability,* P_{alg}.

Lemma 7. *QBUA has a quorum threshold, m, (see Algorithm 2) of* $\lceil \frac{2n}{3} \rceil$ *and can tolerate* $f^s = \frac{n}{3}$ *faulty servers.*

Theorem 8. *QBUA has a better best-effort property than HUA and CUA and a similar best-effort property to ACUA.*

Theorem 9. *QBUA has lower overhead than ACUA and its overhead scales better with the number of node failures.*

Theorem 10. *QBUA limits thrashing by reducing the number of instances of QBUA spawned by concurrent distributed scheduling event.*

6 Experimental Results

We performed a series of simulation experiments on ns-2 to compare the performance of QBUA to ACUA, CUA and HUA in terms of Accrued Utility Ratio (AUR) and Termination-time Meet Ratio (TMR). 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 TMR 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 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.

QBUA 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. We also conducted experiments under a broad range of thread sets. Those results are omitted here due to space constraints; they can be found in [7] and they all exhibit the same trend.

As Figures 1 and 2 show, the performance of QBUA during underloads, in the absence of failure, is similar to that of other algorithms. However, during overloads, QBUA begins to outperform other algorithms due to its better best effort property. During overloads, QBUA accrues, on average, 17% more utility that CUA, 14% more utility than HUA and 8% more utility than ACUA. The maximum difference between the performance of QBUA and other algorithms in our experiment was the 22% difference between ABUA's and CUA's AUR at the

1.88 system load point. Throughout our experiment, the performance of ACUA was the closest to QBUA with the difference in performance between these two algorithms getting more pronounced as system load increases (the largest difference in performance is 11.7% and occurs at about 2.0 system load). The reason for this is that QBUA has a similar best-effort property to ACUA (see Theorem 8). In addition, the difference between these two algorithms becomes more pronounced as system load increases because the scheduling overhead becomes more apparent at high system loads, allowing QBUA, with its lower overhead, to scale better with system load. Also, QBUA does not accrue 100% utility during all cases of underload; as the load approaches 1.0 some deadlines are missed because the overhead of QBUA becomes more significant at this point. This is also true for other collaborative scheduling algorithms such as CUA and ACUA, and, to a lesser extent, for non-collaborative scheduling algorithms such as HUA.

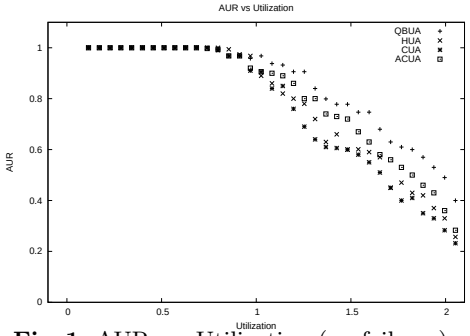

Fig. 1. AUR vs. Utilization (no failures)

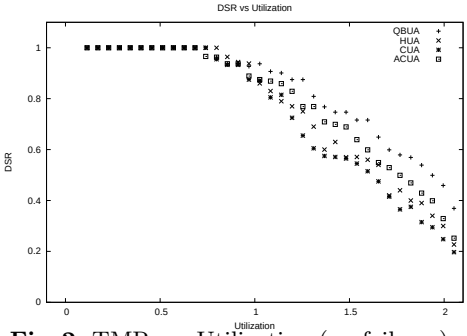

Fig. 2. TMR vs. Utilization (no failures)

Figures 3 and 4 show the effect of failures on QBUA. In these experiments we programmatically fail $f_{max} = 0.2N$ nodes — i.e., we fail 20% of the client nodes. From Figure 3, we see that failures do not degrade the performance of QBUA compared to other scheduling algorithms — i.e., the relationship between the utility accrued by QBUA to the utility accrued by other scheduling algorithms remains relatively the same in the presence of failures. However, QBUA accrues, on average, 18.5% more utility than CUA, 13.6% more utility than HUA and 9.9% more utility than ACUA. Both ACUA and CUA suffer a further loss in performance relative to QBUA in the presence of failures because their time complexity is a function of the number of node failures, therefore they have higher overheads in the presence of failures. In Figure 4 we compare the behavior of QBUA in the presence of failure to its behavior in the absence of failure.

As can be seen, QBUA's performance suffers a degradation in the presence of failures. This degradation is most pronounced during underloads, and becomes less pronounced as the system load is increased. This occurs because, during underloads all threads are feasible and therefore the failure of a node deprives the system of the utility of all the threads that have a section hosted on that node. However, during overloads, not all sections hosted by a node are feasible, thus the failure of that node only deprives the system of the utility of the feasible threads that have a section hosted by that node. Thus the loss of utility during overloads is less than during underloads.

7 Conclusions

We presented a collaborative scheduling algorithm for distributed real-time systems, QBUA. We showed that QBUA has better best-effort properties and message and time complexities than previous distributed scheduling algorithms. We validated our theoretical results using ns-2 simulations. The experiments show that QBUA outperforms other algorithms most during overloads in the presence of failure, due to its better best-effort property and its failure invariant overhead.

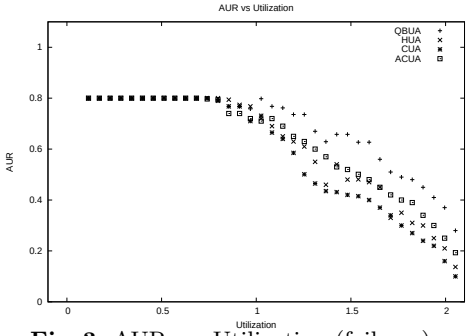

Fig. 3. AUR vs. Utilization (failures)

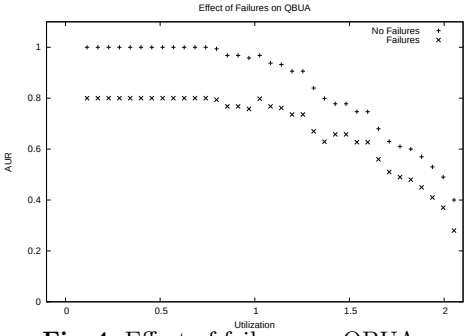

Fig. 4. Effect of failures on QBUA

References

1. Cares, J.R.: Distributed Networked Operations: The Foundations of Network Centric Warfare. iUniverse, Inc. (2006)
2. Clark, R., Jensen, E., Reynolds, F.: An architectural overview of the alpha real-time distributed kernel. In: 1993 Winter USENIX Conf. (1993) 127–146
3. Fahmy, S.F., Ravindran, B., Jensen, E.D.: Scheduling distributable real-time threads in the presence of crash failures and message losses. In: ACM SAC, Track on Real-Time Systems. (2008) To appear, available at: `http://www.real-time.ece.vt.edu/sac08.pdf`.
4. Aguilera, M.K., Lann, G.L., Toueg, S.: On the impact of fast failure detectors on real-time fault-tolerant systems. In: DISC '02, Springer-Verlag (2002) 354–370
5. Chen, W., Toueg, S., Aguilera, M.K.: On the quality of service of failure detectors. IEEE Transactions on Computers **51**(1) (2002) 13–32
6. Hermant, J.F., Lann, G.L.: Fast asynchronous uniform consensus in real-time distributed systems. IEEE Transactions on Computers **51**(8) (2002) 931 – 944
7. Fahmy, S.F., Ravindran, B., Jensen, E.D.: Fast scheduling of distributable real-time threads with assured end-to-end timeliness. Technical report, Virginia Tech, ECE Dept. (2007) Available at: `http://www.real-time.ece.vt.edu/RST_TR.pdf`.
8. Jensen, E., Locke, C., Tokuda, H.: A time driven scheduling model for real-time operating systems (1985) IEEE RTSS, pages 112–122, 1985.
9. Sterzbach, B.: GPS-based clock synchronization in a mobile, distributed real-time system. Real-Time Syst. **12**(1) (1997) 63–75
10. Druschel, P., Rowstron, A.: PAST: A large-scale, persistent peer-to-peer storage utility. In: HOTOS '01. (2001) 75–80
11. Chen, W., Lin, S., Lian, Q., Zhang, Z.: Sigma: A fault-tolerant mutual exclusion algorithm in dynamic distributed systems subject to process crashes and memory losses. In: PRDC '05, Washington, DC, USA, IEEE Computer Society (2005) 7–14
12. Clark, R.K.: Scheduling Dependent Real-Time Activities. PhD thesis, CMU (1990) CMU-CS-90-155.