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

In distributed systems, action and information timeliness is often end-to-end—e.g., a causally dependent, multi-node, sensor to actuator sequential flow of execution in networked embedded systems that control physical processes. Such a causal flow of execution can be caused by a series of nested, remote method invocations. It can also be caused by a series of chained, publication and subscription events, caused due to topical data dependencies—e.g., publication of topic $\mathcal{A}$ depends on subscription of topic $\mathcal{B}$; publication of $\mathcal{B}$, in turn, depends on subscription of topic $\mathcal{C}$, and so on. Designers and users of distributed systems, networked embedded systems in particular, often need to dependably reason about — i.e., specify, manage, and predict — end-to-end timeliness.

Some emerging networked embedded systems are dynamic in the sense that they operate in environments with dynamically uncertain properties (e.g., [1]). These uncertainties include transient and sustained resource overloads (due to context-dependent activity execution times), arbitrary activity arrivals, arbitrary node failures and message loss. Reasoning about end-to-end timeliness is a very difficult and unsolved problem in such dynamic uncertain systems. Another distinguishing feature of motivating applications for this model (e.g., [1]) is their relatively long activity execution time magnitudes—e.g., milliseconds to minutes. Despite the uncertainties, such applications desire the strongest possible assurances on end-to-end activity timeliness behavior.

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 subsequently in Mach 3.0 [3] (a subset), and Real-Time CORBA 1.2 [4] 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.

A distributable thread carries its execution context as it transits node boundaries, including its scheduling parameters (e.g., time constraints, execution time), identity, and security credentials. The propagated thread context is intended to be used by node schedulers for resolving all node-local resource contention among distributable threads such as that for node's physical (e.g., processor, I/O) and logical (e.g., locks) resources, according to a discipline that provides application specific, acceptably optimal, system-wide end-to-end timeliness. Figure 1 shows the execution of four distributable threads. We focus on distributable threads as our end-to-end control flow/programming/scheduling abstraction, and hereafter, refer to them as *threads*, except as necessary for clarity.

Deadlines cannot express both urgency and importance. Thus, we consider the *time/utility function* (or TUF) timeliness model [5] that specifies the utility of completing a thread as a function of that thread's completion time. We specify a deadline as a binary-valued, downward "step" shaped TUF; Figure 2 shows examples. A thread's TUF decouples its importance and urgency—urgency is measured on the X-axis, and importance is denoted (by utility) on the Y-axis.

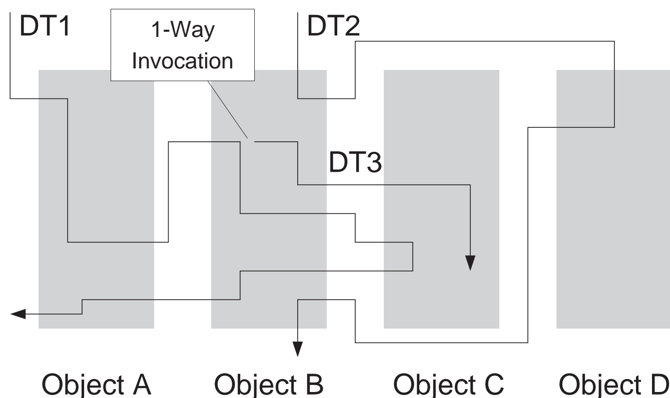

Fig. 1. Four Distributable Threads

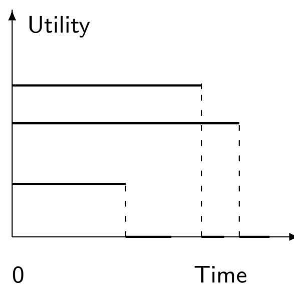

Fig. 2. Example Step TUF Time Constraints

When thread time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued thread utility—e.g., maximizing the total thread accrued utility. Such criteria are called *utility accrual* (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that optimize UA criteria are called UA sequencing algorithms (e.g., [6, 7]).

UA algorithms that maximize total utility under downward step TUFs (e.g., [6, 7]) default to EDF during underloads, since EDF satisfies all deadlines during underloads. Consequently, they obtain the optimum total utility during underloads. During overloads, they inherently favor more important threads over less important ones (since more utility can be attained from the former), irrespective of thread urgency, and thus exhibit adaptive behavior and graceful timeliness degradation. This behavior of UA algorithms is called "best-effort" [6] in the sense that the algorithms strive their best to feasibly complete as many high importance threads — as specified by the application through TUFs — as possible.[3] Consequently, high importance threads that arrive at any time always have a very high likelihood for feasible completion (irrespective of their urgency). Note also that EDF's optimal timeliness behavior is a special-case of UA scheduling.

Contributions. In this work, 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., [2, 8–10]) can be broadly categorized into two classes: *independent node scheduling* and *collaborative scheduling*. In the independent scheduling approach (e.g., [2, 8, 10]), threads are scheduled at nodes using propagated thread scheduling parameters and without any interaction with other nodes (thereby not considering node failures during scheduling). Fault-management is separately addressed by *thread integrity protocols* [11] that run concurrent to thread execution. Thread integrity protocols employ failure detectors (abbreviated here as FDs), and use them to detect failures of the thread abstraction, and to deliver failure-exception notifications [2, 8]. In the collaborative scheduling approach (e.g., [9]), nodes explicitly cooperate to construct system-wide thread schedules, anticipating and detecting node failures using FDs.

There are tradeoffs between the two approaches: Independent node scheduling avoids the overhead of inter-node communication, and is therefore message-efficient (from the thread scheduling standpoint). However, the approach poses theoretical difficulties in establishing end-to-end timing assurances, due to the complex (and concurrent) interaction between thread scheduling and thread fault-management mechanisms. This is overcome

[3] Note that the term "best effort" as used in the context of networks actually is intended to mean "least effort."

in collaborative scheduling, but the approach incurs message overhead costs. In [8, 10], upper bounds are established for such message costs.

FDs that are employed in both paradigms in past efforts have assumed a totally synchronous computational model—e.g., deterministically bounded message delay. While the synchronous model is easily adapted for real-time applications due to the presence of a notion of time, as pointed out in [12], this results in systems with low coverage. On the other hand, it is difficult to design real-time algorithms for the asynchronous model due to its total disregard for timing assumptions. Thus, there have been several (recent) attempts to reconcile these extremes. For example, in [13], Aguilera *et. al.* describe the design of a fast failure detector for synchronous systems and show how it can be used to solve the consensus problem for real-time systems. The algorithm achieves the optimal bound for both message and time complexity for synchronous systems.

In [14], Hermant and Widder describe the *Theta-model*, where only the ratio, Θ, between the fastest and slowest message in transit is known. This increases the coverage of algorithms (designed under this model) as less assumptions are made about the underlying system. While Θ is sufficient for proving the correctness of such algorithms, an upper bound on communication delay is needed to establish timeliness properties.

In [10], Binoy *et. al.* develop HUA, an independent node scheduling algorithm for synchronous systems that uses propagated thread scheduling information to perform local scheduling. This local scheduling sometimes results in a locally optimal decision that may not be globally optimal. In [9], a scheduling algorithm for synchronous systems that uses the collaborative scheduling approach, CUA, is developed. This algorithm uses fast consensus [13] to solve the scheduling problem, but does not provide the same best-effort property as QBUA. In [15], Sherif *et. al.* describe ACUA, a collaborative distributed real-time scheduling algorithm for partially synchronous systems. The algorithm has better best effort properties than previous algorithms for the same problem but has a high overhead due to its reliance on the uniform consensus problem.

In this paper, we target partially synchronous systems that are neither totally synchronous nor totally asynchronous. In particular, we consider the partially synchronous model in [16], where message delay and message loss are probabilistically described. For such a model, we design a thread scheduling algorithm belonging to the collaborative scheduling paradigm, called the *Quorum Based Utility Accrual scheduling* (or QBUA). We show that QBUA satisfies thread time constraints in the presence of crash failures, We also show that the message and time complexity of QBUA compares favorably with other algorithms in its class. Furthermore, we show that QBUA has a better best-effort property — i.e., the affinity for feasibly completing as many high importance threads as possible, irrespective of thread urgency — than past thread scheduling algorithms including [9, 10]. We also prove the exception handling properties of QBUA.

2 Models and Objective

2.1 Models

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., [17]). 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 $\mathbf{T} = \{T_1, T_2, \ldots\}$. The set of sections of a thread T_i is denoted as $[S_1^i, S_2^i, \ldots, S_k^i]$.

Timeliness Model. We specify the time constraint of each thread using a Time/Utility Function (TUF) [5]. 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 $\mathit{\Pi} = \{1, 2, \cdots, n\}$. Bi-directional 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 (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 tasks hosted on the node and a scheduling coprocessor[4]. We also assume that nodes in our system have access to GPS clocks that can provide each node to a UTC clock with high precision [18–20]. Since NCW is our target domain, assuming that each node is equipped with GPS devices is realistic. We also assume that each node is equipped with QoS FDs [16]. The QoS FD descibed in [16] 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 QBUA, and handlers are released on all nodes that host the thread's sections by QBUA. 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*, QBUA 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.

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, [21] — 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 $f_{max} \leq N-1$.

2.2 Scheduling Objectives

Our primary objective is to design a thread scheduling algorithm that will maximize the total utility accrued by all the threads as much as possible. 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 exhibit the best-effort property.

3 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

[4] Dual core technology makes this assumption realistic.

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. 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[5] 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). The details of the algorithm used to compute SWETS can be seen in Algorihtm 5. After computing SWETS, the node contacts those nodes that where affected by arrival of the new distributed scheduling event (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 [22]. 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

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

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
14:                 resp[j] ← (owner, t);
15:             if among resp[], at least m of them are not (nil, nil) then
16:                 if at least m elements in resp[] are (c_1, t) then
17:                     return Compute SWETS;
18:                 else
19:                     if at least m elements in resp[] agree about a certain node then
20:                         for all r_k ∈ Π such that resp[k] ≠ (nil, nil) do
21:                             if resp[k].owner = c_1 then
22:                                 send (RELEASE,timestamp) to r_k;
23:                         Skip rest of algorithm; //Another node is already handling this event
24:                     else
25:                         for all r_k ∈ Π such that resp[k] ≠ (nil, nil) do
26:                             if resp[k].owner = c_1 then
27:                                 send (YIELD,timestamp) to r_k;
28:                             else
29:                                 send (INQUIRE,timestamp) to r_k;
30:                             resp[k] ← (nil, nil);
31:         until forever ;
32:       exit_1:
33:         oldtimestamp ← timestamp;
34:         timestamp ← GetTimeStamp;
35:         for all r_k ∈ Π do
36:             send (RELEASE, oldtimestamp) to r_j;
37:             return;
38: upon receive (CHECK, t) from r_j
39:   if for all instances of QBUA running on this node, timestamp ≠ t then
40:       send (RELEASE, t) to r_j;
41: upon receive (START) from some client node
42:   Update RE_j^i for all sections;
43:   send σ_j and RE_j^i's to requesting node;
```

a sufficient number of replies have arrived (line 15), Algorithm 2 checks whether its request has been accepted by a sufficient ($\lceil \frac{2n}{3} \rceil$ where n is the number of servers, see Section 5) number of server nodes. If so, the node computes SWETS (lines 16-17).

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 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 19-22). The second possibility is that the result of the contention to run QBUA at the servers was inconclusive. This may occur due to differences in communication delay. For example, 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 that the communication delay between each server and client is different. Due to these communication differences, the messages of the clients may arrive in such a patter 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 24-30) and waits for more responses from the server nodes to resolve this conflict. Lines 31-37, release the "lock" on servers after the client node has computed SWETS, lines 38-40 are used to handle the periodic cleanup messages sent by the servers and lines 41-43 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: $\quad CurrentTime \leftarrow$ GetTimeStamp;
7: $\quad$ **if** *(c_1, t') appears in ($c_{owner}[]$,$t_{owner}[]$) or $R_{wait}[]$* **then**
8: $\qquad$ **if** $t < t'$ **then** Skip rest of algo; //This is an old message
9: $\quad$ **if** $tag = REQUEST$ **then**
10: $\qquad$ **if** $\exists\, t_{grant} \in t_{grant}[]$ *such that* $t \leq t_{grant}$ **then**
11: $\qquad\quad$ **send** (RESPONSE, c, t_{grant}) to c_1; //where $c \leftarrow c_{owner}[i]$, such that $t_{grant}[i] = t_{grant}$;
12: $\qquad\quad$ Enqueue (c_1, t) in $R_{wait}[i]$, such that $t_{grant}[i] = t_{grant}$;
13: $\qquad\quad$ Skip rest of algorithm;
14: $\qquad$ **else**
15: $\qquad\quad$ AddElement($c_{owner}[], c_1$);
16: $\qquad\quad$ AddElement($t_{owner}[], t$);
17: $\qquad\quad$ AddElement($t_{grant}[], CurrentTime$);
18: $\qquad\quad$ **send** (RESPONSE, c_1, t) to c_1;
19: $\quad$ **else if** $tag = RELEASE$ **then**
20: $\qquad$ Delete entry corresponding to c_1, t from $c_{owner}[]$, $t_{owner}[]$, $t_{grant}[]$, and $R_{wait}[]$;
21: $\quad$ **else if** $tag = YIELD$ **then**
22: $\qquad$ **if** $(c_1,\ t) \in (c_{owner}[],\ t_{owner}[])$ **then**
23: $\qquad\quad$ For i, such that $(c_1,\ t) = (c_{owner}[i],\ t_{owner}[i])$
24: $\qquad\quad$ Enqueue (c_1, t) in $R_{wait}[i]$;
25: $\qquad\quad$ $(c_{wait}, t_{wait}) \leftarrow$ top of $R_{wait}[i]$;
26: $\qquad\quad$ $c_{owner}[i] \leftarrow c_{wait}$; $t_{owner}[i] \leftarrow t_{wait}$;
27: $\qquad\quad$ $t_{grant}[i] \leftarrow CurrentTime$;
28: $\qquad\quad$ **send** (RESPONSE, c_{wait}, t_{wait}) to c_{wait};
29: $\qquad$ **if** $c_1 \notin c_{owner}[]$ **then**
30: $\qquad\quad$ $(c, t_p) \leftarrow (c_{owner}[i], t_{owner}[i])$, for min i such that $t \leq t_{grant}[i]$;
31: $\qquad\quad$ **send** (RESPONSE, c, t_p) to c_1;
32: $\quad$ **else if** $tag = INQUIRE$ **then**
33: $\qquad$ $(c, t_p) \leftarrow (c_{owner}[i], t_{owner}[i])$, for min i such that $t \leq t_{grant}[i]$;
34: $\qquad$ **send** (RESPONSE, c, t_p) to c_1;
35: **upon** suspect that $c_{owner}[i]$ has failed:
36: $\quad$ HandleFailure($c_{owner}[i]$,$c_{owner}[]$, $t_{owner}[]$,$t_{grant}[]$,$R_{wait}[]$);
37: **periodically**:
38: $\quad \forall\, c_{owner} \in c_{owner}[]$:
39: $\qquad$ **send** (CHECK, t_{owner}) to c_{owner}; //NB. t_{owner} is the entry in $t_{owner}[]$ that corresponds to c_{owner}.

We now turn our attention to the algorithm run by the servers (Algorithm 3). 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

Algorithm 4: HandleFailure($c_{owner}[i]$,$c_{owner}[]$, $t_{owner}[]$,$t_{grant}[]$,$R_{wait}[]$)

```
1: if R_wait[i] is empty then
2:     remove c_owner[i]'s entry from c_owner[], t_owner[], t_grant[];
3:     Delete R_wait[i];
4: else
5:     CurrentTime ← GetTimeStamp;
6:     (c_wait, t_wait) ← top of R_wait[i];
7:     c_owner[i] ← c_wait; t_owner[i] ← t_wait;
8:     t_grant[i] ← CurrentTime;
9:     send (RESPONSE, c_wait, t_wait) to c_wait;
```

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 (which is 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 (lines 7-8).

Starting at line 9, the algorithm begins to examine the message it has received from a client node. 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 (lines 10-18). If there is indeed such an instance, 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 (line 11-13).

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). Therefore, we guarantee that this contention will be resolved in the second round of the algorithm. The rest of the algorithm shows the response of servers to INQUIRE messages (lines 32-34) and clean up procedures that remove stale messages from the server (lines 35-39). As can be seen on line 36, when a node that is currently running an instance of QBUA fails, HandleFailure($c_{owner}[i]$,$c_{owner}[]$, $t_{owner}[]$,$t_{grant}[]$,$R_{wait}[]$) is called to handle this failure. Algorithm 4 shows the details of this function. If the waiting queue corresponding to this instance of QBUA, $R_{wait}[i]$, is empty, then there are no other nodes that have detected the event that triggered QBUA on $c_{owner}[i]$ and so the system is cleared of this instance of QBUA (lines 1-3). Otherwise, there are other nodes that have detected the event that triggered QBUA on $c_{owner}[i]$, or another concurrent event, and therefore the failure of $c_{owner}[i]$ results in selecting another node from the waiting queue $R_{wait}[i]$ to run QBUA to handle this event (lines 4-9).

We now turn our attention to the algorithm used by a client node to compute SWETS once it has received information from all other nodes in the system. In order to compute SWETS, a node needs to perform 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 these threads, in non-increasing order of global PUD, into the scheduling queues of all nodes in the system. After inserting the sections into the scheduling queues, the node checks whether the current schedule is feasible. If it is not feasible, then the thread is removed from SWETS (after scheduling the appropriate exception handler if the thread has already executed some of its sections).

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 [7]. 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 be able to complete their execution before their assigned deadline. Since we are considering threads with end-to-end deadlines, we now turn our attention to deriving the deadlines of the component sections of these threads. The termination time of each section belonging to a thread needs to be derived from that 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:

$$S^i_j.tt = \begin{cases} T_i.tt & j = k \\ S^i_{j+1}.tt - S^i_{j+1}.ex - T & 1 \leq j \leq k-1 \end{cases}$$

where $S^i_j.tt$ denotes section S^i_j's termination time, $T_i.tt$ denotes T_i's termination time, and $S^i_j.ex$ denotes the estimated execution time of section S^i_j. 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 deadline with a certain, high, probability. Using these derived termination times, we can determine the feasibility of a schedule.

In addition, each handler has a **relative** termination time, $S^h_j.X$. 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:

$$S^h_j.tt = \begin{cases} S^i_k.tt + S^h_j.X + T_D + t_a & j = k \\ S^h_{j+1}.tt + S^h_j.X + T & 1 \leq j \leq k-1 \end{cases}$$

where $S^h_j.tt$ denotes section handler S^h_j's termination time, $S^h_j.X$ denotes the relative termination time of section handler S^h_j, $S^i_k.tt$ 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. From this termination time decomposition, we compute start times for each handler:

$$S^h_j.st = \left\{ S^h_j.tt - S^h_j.ex \quad 1 \leq j \leq k \right.$$

where $S^h_j.ex$ denotes the estimated execution time of section handler S^h_j. Thus, we assure the feasible execution of the exception handlers of failed sections, in order to revert the system to a safe state.

Algorithm 5 contains the details of our scheduling algorithm. Once a node has received information from all other nodes in the system (line 2 in Algorithm 1), it can begin to compute SWETS. Each node, j, sends the node running QBUA its current local schedule σ^p_j. Using these schedules, the node can determine the set of threads, Γ, that are currently in the system. Both these variables are used as input to the scheduling algorithm (lines 1 and 2 in Algorithm 5). In lines 3-7, the algorithm computes the global PUD of each thread in Γ using the method we describe above.

The next step would be to schedule the threads in non-increasing order of their PUD. However, 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 deadline. 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 8-11). Since these handlers were part of σ^p_j, and our algorithm always maintains the feasibility of a schedule as an algorithm invariant, then we are sure that these handlers will be able to execute to completion before their deadline.

In line 12, we sort the threads in the system in non-increasing order of PUD. We then consider the threads for scheduling in that order (lines 13-26). As can be seen in those lines, we consider each remaining section of the thread for scheduling. In lines 15-16 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 (i.e. its

Algorithm 5: ConstructSchedule

1: **input:** Γ; //Set of threads in the system
2: **input:** σ_j^p, $H_j \leftarrow$ nil; //σ_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 *belonging to* T_i, $t_{curr} + S_j^i.ex > S_j^i.tt$ **then**
5:         $T_i.PUD \leftarrow 0$;
6:     **else**
7:         $T_i.PUD \leftarrow \frac{U_i(t_{curr}+GE_i)}{GE_i}$;
8: **for** *each task* $el \in \sigma_j^p$ **do**
9:     **if** *el is an exception handler for section* S_j^i **then**
10:         Insert(el, H_j, $el.tt$);
11: $\sigma_j \leftarrow H_j$;
12: $\sigma_{temp} \leftarrow$ sortByPUD(Γ);
13: **for** *each* $T_i \in \sigma_{temp}$ **do**
14:     $T_i.stop \leftarrow false$;
15:     **if** *did not receive* σ_j *from node hosting one of* T_i*'s sections* S_j^i **then**
16:         $T_i.stop \leftarrow true$;
17:     **for** *each remaining section,* S_j^i, *belonging to* T_i **do**
18:         **if** $T_i.PUD > 0$ *and* $T_i.stop \neq true$ **then**
19:             Insert(S_j^i, σ_j, $S_j^i.tt$);
20:             **if** $S_j^h \notin \sigma_j^p$ **then**
21:                 Insert(S_j^h, σ_j, $S_j^h.tt$);
22:             **if** *isFeasible*(σ_j)=*false* **then**
23:                 $T_i.stop \leftarrow true$;
24:                 Remove(S_k^i, σ_k, $S_k^i.tt$) for $1 \leq k \leq j$;
25:                 **if** $S_j^i \notin \sigma_j^p$ **then**
26:                     Remove(S_j^h, σ_j, $S_j^h.tt$);
27: **for** *each* $j \in N$ **do**
28:     **if** $\sigma_j \neq \sigma_j^p$ **then**
29:         Mark node j as being affected by current scheduling event;

deadline hasn't passed) and the thread has not been rejected from the system, then we insert that section into the scheduling queue of the node responsible for it (lines 18-19).

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 8-11. Otherwise, the handler is inserted into its corresponding ready queue (lines 20-21). Once the section, and its handler, have been inserted into the ready queue, we check the feasibility of the schedule (line 22). If the schedule is not feasible, we remove the thread's sections from the schedule (line 24). Notice that if one section belonging to a thread cannot be scheduled, all predecessor sections of the thread accepted into the system are removed (line 24). This ensures that a thread that cannot complete does not take up the valuable computation resources on any node. However, we first check to see whether the section's handler was part of a previous schedule before we remove it (lines 25-26). The reason we perform this check before removing the handler of the section is that if the handler was part of a previous schedule, then that 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 27-29). It is to these nodes that the current node needs to multicast the changes that have occurred (line 4, Algorithm 1).

We now turn our attention to the last piece of the puzzle. We use a function *isFeasible* in Algorithm 5 to determine the feasibility of the schedules we construct. Algorithm 6 contains the details of this algorithm. In order to determine whether a schedule is feasible or not, we need to check if all the sections in the schedule can complete before their derived termination times.

The loop on lines 3-6 examines each section in the schedule and attempts to determine whether or not it can complete before its termination time. Since QBUA is executed before sections have actually arrived at some of the nodes in the system, we need to provide an estimate for the starting time of each section on a node. A section can start immediately when it arrives at a node or it may wait some time if other sections are scheduled for execution before it. Therefore, the start time of a section is the maximum of the termination time of the last section scheduled to execute before it ($CumExeTime$) and the time it arrives at the node, which

we estimate using the finish time of its predecessor section, plus the communication delay $S^i_{j-1}.ff + T$. Note that this finish time, $S^i_{j-1}.ff$, is different from the termination time of a section's predecesor, $S^i_{j-1}.tt$. While the former is an estimate of the time when the predecessor section will finish on the node that hosts it, the latter is the worst case finish time that a predecessor section needs to meet in order for its successor sections to be seduceable. To this start time, we add the execution time of the section, $S^i_j.ex$, to obtain an estimate of the expected completion time of the section (line 4). We then check if this value is greater than the derived termination times of the section; if so, the schedule is infeasible (lines 6-7).

Algorithm 6: *isFeasible*

```
1: input: σ;output: true or false;
2: Initialization: CumExeTime=t_curr;
3: for each section S^i_j ∈ σ do
4:     CumExeTime ← max(CumExeTime, S^i_{j-1}.ff + T) + S^i_j.ex; // If j=1, then S^i_{j-1}.ff = 0 and T = 0;
5:     S^i_j.ff ← CumExeTime;
6:     if CumExeTime > S^i_j.tt then return false;
7: return true;
```

QBUA's dispatcher is shown in Algorithm 7. As can be seen, 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 by a node. The events and the actions taken in response to these events are self explanatory.

Algorithm 7: Event Dispatcher on each node i

```
1  Data: schedevent, current schedule σ_p;
2  switch schedevent do
3      case invocation arrives for S^i_j
4          mark segment S^i_j ready;
5      case segment S^i_j completes
6          remove S^i_j from σ_r,σ_p;
7          remove S^h_j from H;
8          set RE^i_j to zero;
9      case S^h_j ∈ H and S^h_i.st expires
10         mark handler S^h_j ready;
11     case downstream handler S^h_{j+1} completes
12         mark handler S^h_j ready;
13     case handler S^h_j completes
14         remove S^h_j from σ_p,H;
15         notify scheduler for S^h_{j-1};
16     case new thread, T_i, arrives
17         if origin node, send segments S^i_j to all;
18         pass event to QBUA;
19     case node failure detected
20         pass event to QBUA;
21 execute first ready segment in σ_p;
```

5 Algorithm Properties

We now turn our attention to proving some theoretical results for the algorithm. 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}.*

Proof. When a distributed scheduling event is detected by a node, it contacts the quorum system to determine whether or not to start an instance of QBUA (see Section 4). T time units is used to contact the quorum nodes,

and another T time units is taken for the reply of the quorum nodes to reach the requesting node. After these two communication steps, two outcomes are possible.

One possibility is that a quorum ($\frac{2n}{3}$) of servers receive a particular node's request first and so grant that node permission to run an instance of QBUA (lines 15-23, Algorithm 2). In this case, only $2T$ time units are necessary to come to a decision.

The second possibility is that none of the client nodes receive permission from a quorum of server nodes, and therefore the result of the first round of contention is inconclusive (see Section 4 for an example). In this case, all nodes send YIELD messages to the server nodes to relinquish the "lock" they where granted to run an instance of QBUA (lines 24-30, Algorithm 2). As discussed in Section 4, the above scenario is caused by differences in communication delays between different client-server pairs. However, by the time that the YIELD messages reach the server nodes ($3T$), all client requests must have reached the server nodes and will be present in their waiting queue (line 12, Algorithm 3) so the server nodes can now make a decision about which node gets to run QBUA (lines 22-31, Algorithm 3) by selecting the earliest request in its waiting queue (ties are broken using client ID). Therefore, the contention is resolved in $4T$ messages delays, one T for each of the REQUEST, RESPONSE, YIELD and RESPONSE messages communicated between client server pairs.

Thus, the whole process of using the quorum system to determine whether or not to run an instance of QBUA takes $4T$ time units in the worst case. Since each of the communication delays, T, are random variables with CDF $DELAY(t)$. The probability that a communication round will take more than T time units is $p = 1 - DELAY(T)$. Since there are four communication rounds, the probability that none of these rounds take more than T time units is $P = bino(0, 4, p)$, where $bino(x, n, p)$ is the binomial distribution with parameters n and p. Thus the probability that a node determines whether or not it needs to run a version of QBUA after $4T$ is also $P_{lock} = bino(0, 4, p)$. □

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}.*

Proof. Once a node is granted permission to run an instance of QBUA it executes Algorithm 1. 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 1 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 take $|\Gamma|k$ time units for threads with k sections each. The for loop on lines 8-10 will take wN time units to examine the w sections in the scheduling queue of each of the N nodes in the system. Line 12 takes $O(|\Gamma| \log(|\Gamma|))$ time units to sort the threads in non-increasing order of global PUD using quick sort. The two nested loops on lines 13-26 take $|\Gamma|k^2w$ in the worst case, since there are $|\Gamma|$ threads each with k sections to insert into scheduling queues and each queue needs to be tested for feasibility after the insertion of a section using the linear time function $isFeasible$ in $O(w)$ time and removing all previously accepted sections, line 24, can take at most O(k) time. Finally, lines 27-29 determines which nodes need to be notified of changes in $O(N)$ time.

Thus the total time complexity of the algorithm is $3T + |\Gamma|k + wN + O(|\Gamma| \log(|\Gamma|)) + |\Gamma|k^2w + O(N)$. If we consider the number of sections in a thread, k, and the number of sections in the waiting queue of a node, w, to be constants, then the asymptotic time complexity of the algorithm is $O(T + N + |\Gamma| \log(|\Gamma|))$.

There are three communication rounds in this procedure. However, the first two of these communication rounds depend on timeouts (line 2, Algorithm 1), 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 1 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 $O(T + N + |\Gamma| \log(|\Gamma|) + T_D)$ time units after it occurs, with high, computable, probability, P_{hand}.*

Proof. There are two possible distributed scheduling events: 1) the arrival of a new thread into the system and 2) the failure of a node.

In case of the arrival of a new thread, the root node of that thread immediately attempts to acquire a "lock" on running an instance of QBUA. By Lemma 1, the node takes $4T$ time units to acquire a lock and by Lemma 2, it takes the algorithm $O(T + N + |\Gamma| \log(|\Gamma|))$ to compute SWETS. Therefore, in the case of the arrival of a

thread the event is handled $O(T+N+|\Gamma|\log(|\Gamma|)+4T)$=$O(T+N+|\Gamma|\log(|\Gamma|))$ time units after it occurs. Note that $O(T+N+|\Gamma|\log(|\Gamma|))$ is $O(T+N+|\Gamma|\log(|\Gamma|)+T_D)$.

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 QBUA. By Lemmas 1 and 2, this takes $O(T+N+|\Gamma|\log(|\Gamma|))$ time units. Thus the event is handled $O(T+N+|\Gamma|\log(|\Gamma|)+T_D)$ 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 $P_{hand} = P_{SWETS} \times P_{lock}$. □

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

Proof. The actual message cost of the algorithm is $5n+3N$. The $5n$ component of the message complexity comes from the quorum based arbitration system used in the algorithm. The $5n$ comes from n messages for REQUEST, RESPONSE, YIELD/INQUIRE, RESPONSE and RELEASE respectively. After a node has acquire a "lock", it broadcasts a start message (line 1, Algorithm 1) this takes N messages. The nodes then reply to the current node (line 2, Algorithm 1) using another N messages. Finally, the current node multicasts its results to affected nodes (line 4, Algorithm 1) using another N messages (because in the worst case all nodes in the system may be affected). Thus the actual message complexity of the algorithm is $5n+3N$ which is asymptotically $O(n+N)$. □

Note that this message complexity is much better than the message complexity of consensus based algorithms (which tend to be $O(N^2)$ because every node communicates with every other node). Thus our algorithm is more scalable in that sense. It should also be noted that n (the number of servers in the system) is much smaller than the total number of nodes in the system, N.

Lemma 5. *If all nodes are underloaded and no nodes fail, then no threads will be suggested for rejection by QBUA 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 1). This can occur due to one of two reasons; 1) the broadcast message (line 1, Algorithm 1), 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 QBUA 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 $P_{tmp} = bino(0, N, p)$ where $bino(x, n, p)$ 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 $tmp = bino(0, N, p)$. 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, $p_{norej} = tmp \times P_{tmp}$. □

Lemma 6. *If each section of a thread meets its derived termination time, then under QBUA, 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 $p_{suc} = bino(0, k, p)$. Therefore, the probability that the thread meets its termination time is also $p_{suc} = bino(0, k, p)$. □

Theorem 7. *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}.*

Proof. By Lemma 5, 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 6, this implies that each thread, j, will meet its termination time with probability p^j_{suc}. Therefore, for a system with $X = |\Gamma|$ threads, the probability that all threads meet their termination time is $P_{tmp} = \prod_{j=1}^{X} p^j_{suc}$. 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 $O(T + N + |\Gamma| \log(|\Gamma|))$ time units and still be schedulable because QBUA takes $O(T+N+|\Gamma| \log(|\Gamma|))$ 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. □

Theorem 8. *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}.*

Proof. As in Lemma 5, 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 $bino(0, N - f, p)$ where $p = 1 - DELAY(T)$. Therefore, the probability that none of the threads in Γ are rejected is $P_{norej} = bino(0, N-f, p) \times bino(0, N-f, p)$. This means that all the sections belonging to those threads will be scheduled to meet their derived termination times. By Lemma 6, this implies that each of these threads, T_j, will meet their termination times with probability p^j_{suc}. 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 $P_{tmp} = \prod_{j \in \Gamma} p^j_{suc}$. The probability that all the remaining threads are execute to completion is thus $P_{alg} = P_{tmp} \times p_{norej}$. □

Lemma 9. *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.*

Proof. Our algorithm considers a memoryless crash recovery model for the quorum nodes. This means that a quorum node that crashes and then recovers losses all its state information and starts from scratch. What this implies is that for our algorithm to tolerate such failures, the threshold m should be large enough such that there is at least one correct server in the intersection of any two quorums.

Assume that f is the maximum number of faulty servers in the system (i.e. servers that may fail at some time in the future), then the above requirement can be expresses as $2m - n > f^s$. On the other hand, m cannot be too large since some servers will fail and choosing too large a value of m may mean that client nodes may wait indefinitely for responses from servers that have failed. The requirement translates to $m \leq n - f^s$. Combining the two we get, $f^s = \frac{n}{3}$ and m can be set to $\lceil \frac{2n}{3} \rceil$. □

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

Lemma 10. *If a node hosting a section, S^i_j, 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 QBUA 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 [16], all nodes in the system will detect the failure of the node at time $t_f + T_D$. As a result, the QBUA algorithm will be triggered and will exclude T_i from the system because node j will not send its schedule (lines 15-16 Algorithm 5). Consequently, Algorithm 5 will include the section handlers for this thread in H. Execution of QBUA completes in time t_a and thus all handlers will be included in H 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}. □

Lemma 11. *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 QBUA, its handler S^h_i will be released no earlier than S^h_{i+1}'s completion and no later than $S^h_{i+1}.tt + T + S^h_i.X - S^h_i.ex$.*

Proof. For $i \neq k$, a section's exception handler can be released due to one of two events; 1) its start time expires (lines 9-10 in Algorithm 7); or 2) an explicit invocation is made by the handler's successor (lines 11-12 in Algorithm 7).

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. □

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 12. *If a section S_i fails (per Definition 1), then under QBUA, 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 18-26 of Algorithm 5).

When some previous head node of S_i's thread fails, QBUA 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 8-11 and lines 18-26). Since the schedule is feasible and S_i^h is inserted to meet $S_i^h.tt$ (line 10), then S_i^h will complete by time $S_i^h.tt$. □

We now state QBUA's bounded clean-up property.

Theorem 13. *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_{exep}.*

Proof. The LIFO property follows from Lemma 11. 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 11), then we guarantee LIFO execution of the handlers.

The fact that all correct handlers complete in bounded time is shown in Lemma 12, 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 10) with probability P_{hand} and QBUA guarantees that all these sections will complete before their termination times (Lemma 12). 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 QBUA to execute).

Since Lemma 12 guarantees that all handlers will finish before their derived termination times, the only stochastic part of the theorem is the probability that QBUA will include the handlers of all the section in time $t_f + T_D + t_a$. From Lemma 10, we know this probability is P_{hand}, thus $P_{exep} = P_{hand}$. □

Definition 2. Consider a distributed scheduling algorithm $\mathcal{A}$. DBE is defined as the property that $\mathcal{A}$ orders its threads in non-increasing order of global PUD while considering them for scheduling and schedules all feasible threads in the system in that order. The global PUD of a thread is the ratio of the utility of the thread to the sum of the remaining execution times of all its sections.

Note that the *DBE* property is essential for any UA algorithm that attempts to maximize system wide accrued utility by favoring tasks that offer the most utility for the least amount of execution time (which is the heuristic used by DASA [7]). For the proofs of Lemmas 14, 15 and 16, please see [15].

Lemma 14. *HUA [10], does not have the DBE property.*

Lemma 15. *CUA [9] does not have the DBE property.*

Lemma 16. *ACUA has the DBE property for threads that can survive the scheduling overhead of the algorithm — i.e., threads that can be delayed $O(fT + Nk)$ and still be schedulable.*

Lemma 17. *QBUA has the DBE property for threads that can survive the scheduling overhead of the algorithm — i.e., threads that can be delayed $O(T + N + |\Gamma| \log(|\Gamma|))$ (see Theorem 3) and still be schedulable.*

Proof. The DBE property requires all threads to be ordered in non-decreasing order of global PUD, and this is accomplished in lines 3-7 and line 12 of Algorithm 5. In addition, the DBE property requires that all feasible threads be scheduled in non-decreasing order of PUD, and this is accomplished in lines 13-26 of Algorithm 5. Thus QBUA has the DBE property for all threads that can withstand the $O(T + N + |\Gamma| \log(|\Gamma|))$ overhead of the algorithm and still remain schedulable. □

Theorem 18. *QBUA has a better best-effort property than HUA and CUA and a similar best-effort property to ACUA.*

Proof. The proof follows directly from Lemmas 14, 15, 16 and 17. In particular, HUA and CUA do not have the DBE property while QBUA does, and both QBUA and ACUA have the DBE property but for threads that can survive their, different, scheduling overheads. □

Lemma 19. *The message overhead of QBUA is better than the message overhead of ACUA and scales better with the number of node failures.*

Proof. The message complexity of ACUA is $O(fN^2)$ which is clearly asymptotically more expensive than the O($n+N$) message complexity of QBUA. In addition, since the message complexity of ACUA is a linear function of f, the number of failed nodes, and the message complexity of QBUA does not depend on f —i.e., is not affected by the number of node failures— the message overhead of QBUA scales better in the presence of failure. □

Lemma 20. *The time overhead of QBUA is asymptotically similar to the time overhead of ACUA and scales better with the number of node failures. In addition, when the number of threads in the system is fixed, the time complexity of QBUA is asymptotically better than that of ACUA and scales better in the presence of failure.*

Proof. The time complexities of the two algorithms, $O(T + N + |\Gamma| \log(|\Gamma|))$ for QBUA and $O(fT + kN)$ for ACUA, are asymptotically similar, but since the time complexity of ACUA is a function of f and QBUA's is not, QBUA's time complexity scales better in the presence of failure.

When the number of threads in the system is fixed, the term $|\Gamma| \log(|\Gamma|)$ in the time complexity of QBUA becomes a constant and thus its asymptotic time complexity becomes $O(T + N)$ which is better than the time complexity of ACUA, $O(fT + kN)$. Further, since the time complexity of QBUA is not a function of f and the time complexity of ACUA is, QBUA's complexity scales better in the presence of failure. □

Theorem 21. *QBUA has lower overhead than ACUA and its overhead scales better with the number of node failures.*

Proof. The proof follows directly from Lemmas 19 and 20. □

In should be noted that in our computation of time complexity of algorithms, we do not take into account the effect of message overhead. However, the message complexity affects the utilization of the communication channel and the queue delay at nodes and hence has a direct impact on communication delay (which appears as a term in both time complexities mentioned above). The effect of this higher message complexity can be seen in our experiments (see Section 6), where the higher overhead of ACUA, both message and time overheads, results in performance that is worse than that of QBUA.

Theorem 22. *QBUA limits thrashing by reducing the number of instances of QBUA spawned by concurrent distributed scheduling event.*

Proof. Thrashing occurs when concurrent distributed events spawn, superfluous, separate instances of QBUA. QBUA prevents this by having nodes wishing to run an instance of QBUA contact a quorum system to gain permission for doing so (see Section 4). In lines 9-13 of Algorithm 3, the quorum system does not spawn a new instance of QBUA if there is an instance of QBUA already running that was granted permission to start *after* the timestamp of the arriving scheduling event. This occurs because the instance of QBUA that started after the scheduling event occurred will have information about that event and will thus handle it. This reduces thrashing by prevent superfluous concurrent instances from running at the same time. □

6 Experimental Results

We performed a series of simulation experiments on ns-2 [23] 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%. In order to make the comparison fair, all the 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 all the experiments we perform, the utilization of the system is considered the *maximum* utilization experienced by any node. While conducting our experiments, we considered three different thread sets.

Thread set I. In the first thread set we consider, our thread set parameters — i.e., section execution times, thread termination times, and thread utility — are chosen to highlight the better distributed best-effort properties of QBUA. The strength of QBUA lies in its ability to give priority to threads that will result in the most system-wide accrued utility. Therefore, the thread set that highlights this property is one that contains threads that would be given low priority on a node if local scheduling is performed but should be assigned high priority due to the system-wide utility that they accrue to the system. Therefore, our first thread set 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 section). 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.

Thread set II. In the second thread set, the section execution times, thread utilities and termination times are generated using uniform random variables. This thread set represents the random load that a system may experience during normal execution. Using this thread set, we compare the performance of the algorithms to see how they respond to "normal" thread sets.

Thread set III. Finally, since the second threat set is neutral to all algorithms in the sense that it is randomly generated and does not fit the performance of any particular algorithm better than the other, and the first thread set favors QBUA since it specifically contains thread sets that are best scheduled by collaborative algorithms to avoid "local minumums", our last thread set is designed to contain threads that can be scheduled to accrue maximum utility without requiring collaboration. Our intention in choosing such a thread set is to see how QBUA compares to other algorithms when collaboration is not required.

Figures 3 and 4 show the result of our AUR and DSR experiments in the absence of node failure for thread set I. As Figures 3 and 4 show, the performance of QBUA during underloads is similar to that of other distributed real-time scheduling 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 the 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 behavior is that QBUA has a similar best-effort property to ACUA (see Theorem 18). In addition, we believe that the difference between these two algorithms becomes more pronounced as system load

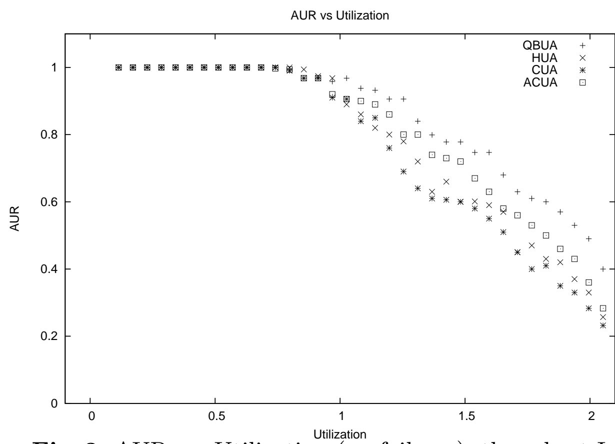

Fig. 3. AUR vs. Utilization (no failures), thread set I

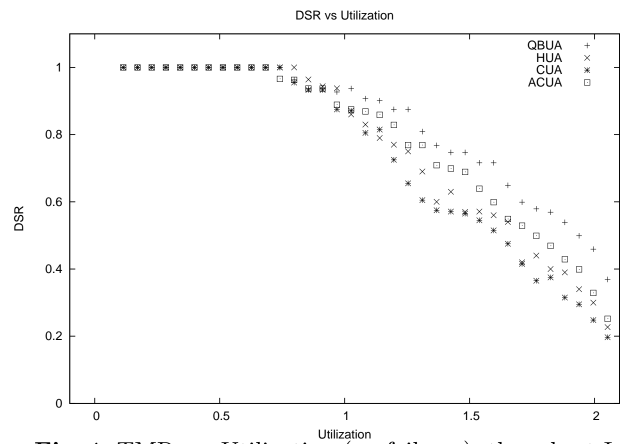

Fig. 4. TMR vs. Utilization (no failures), thread set I

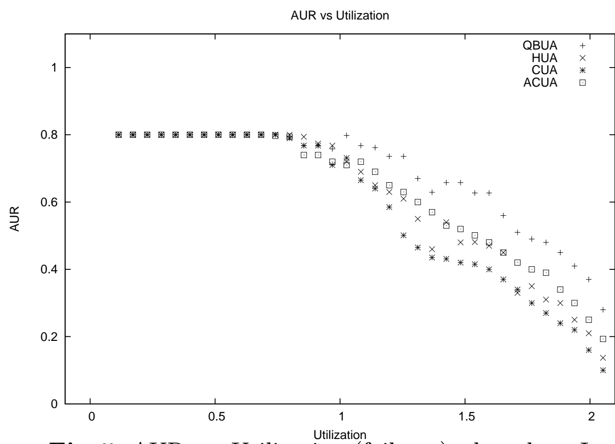

Fig. 5. AUR vs. Utilization (failures), thread set I

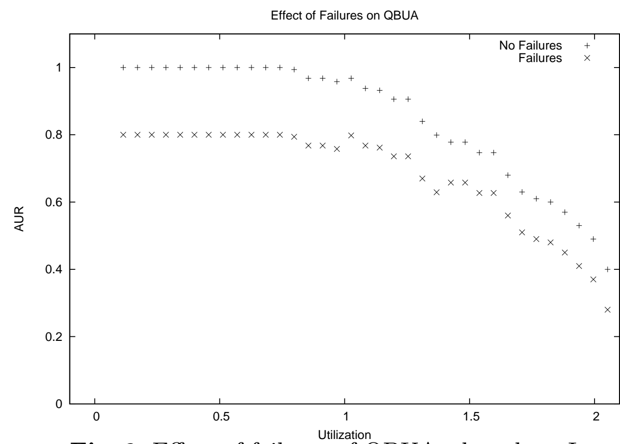

Fig. 6. Effect of failures of QBUA, thread set I

increases because the delay caused by the scheduling overhead has greater consequences on the schedulability of the system due to the decreased system slack during overloads. Thus QBUA's lower overhead allows it to scale better with system load. Another interesting aspect of the experiment is that, contrary to Theorem 7, QBUA does not accrue 100% utility during all cases of underload. As the load on the system 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 is true to a lesser extent for non-collaborative scheduling algorithms such as HUA due to their lower overhead.

Figures 5 and 6 show the effect of failures on QBUA when thread set I is used. In these experiments we programmatically fail $f_{max} = 0.2N$ nodes — i.e., we fail 20% of the client nodes. From Figure 5, 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, it is interesting to note that now QBUA accrues, on average, 18.5% more utility than CUA, 13.6% more utility than HUA and 9.9% more utility than ACUA. It should be noticed that both ACUA and CUA suffer a further loss in performance relative to QBUA in the presence of failures. The main reason this occurs is that both these algorithms have a time complexity that is a function of the number of node failures, therefore they have higher overheads in the presence of failures and this affects their results.

In Figure 6 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. However, from Theorem 8, we know that QBUA meets the termination times of all threads hosted by surviving nodes.

As can be seen from the figure, the difference in performance of QBUA in the presence of failure is most pronounced during underloads, and becomes less pronounced as the system load is increased. The reason for this is that during underloads all threads are feasible and therefore the failure of a particular 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. This amount is less than would occur if the system were deprived of the utility of all threads hosted on the node and leads to a less pronounced effect on the performance of QBUA in the presence of failures.

In Figures 7, 8, 14 and 10, the same experiments as above are repeated using thread set II. As can be seen, the results follow a similar pattern to the experiments for thread set I, but now the difference between QBUA and other algorithms is not so pronounced. This occurs because the randomly generated thread set is likely to contain some threads that need collaborative scheduling but not as many as in the thread set that we specifically designed to contain such threads.

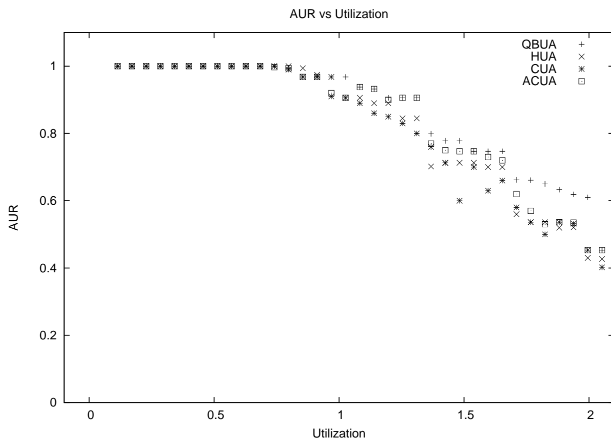

Fig. 7. AUR vs. Utilization (no failures), thread set II

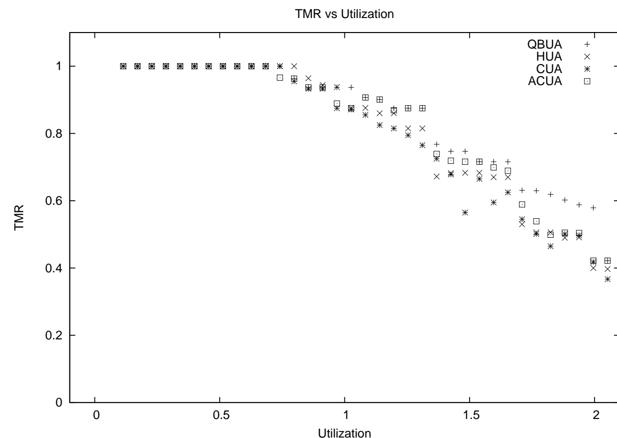

Fig. 8. TMR vs. Utilization (no failures), thread set II

In Figures 11, 12, 13 and 14, the same experiments as above are repeated using thread set III. In these experiments, the best performing algorithm is HUA since it has the least overhead (it does not perform any collaborative scheduling). The performance of the other three algorithms are similar to each other with the

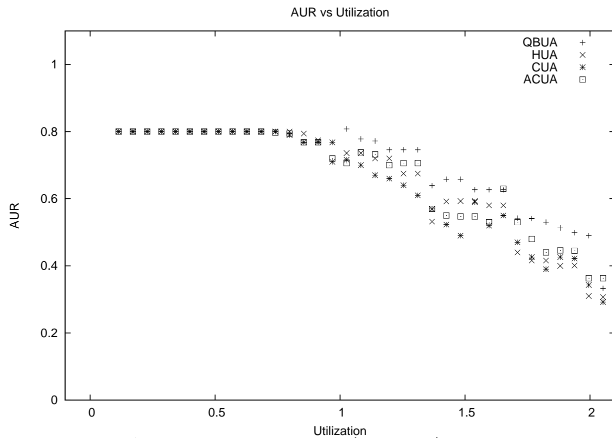

Fig. 9. AUR vs. Utilization (failures), thread set II

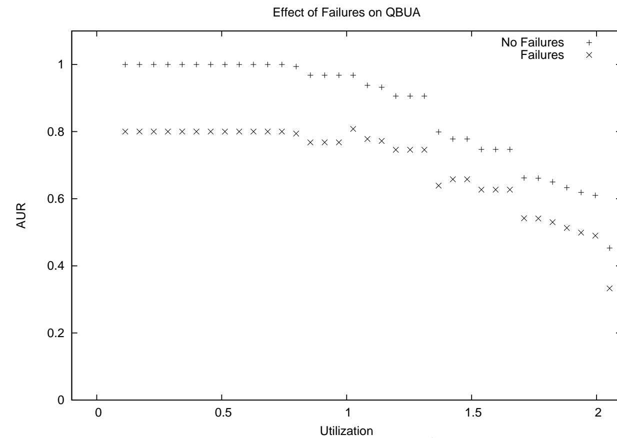

Fig. 10. Effect of failures of QBUA, thread set II

performance of the consensus based algorithms (CUA and ACUA) being slightly worse than that of QBUA since they have higher overhead.

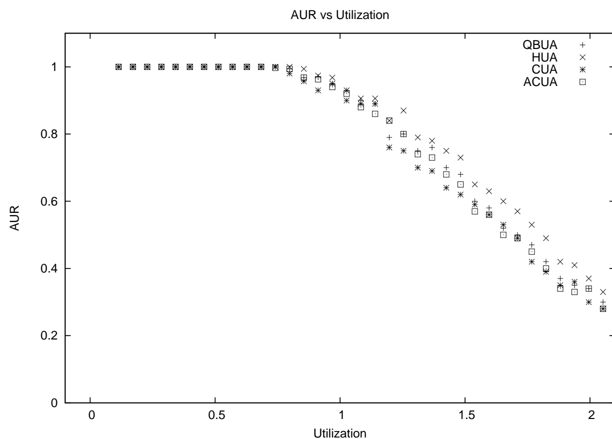

Fig. 11. AUR vs. Utilization (no failures), thread set III

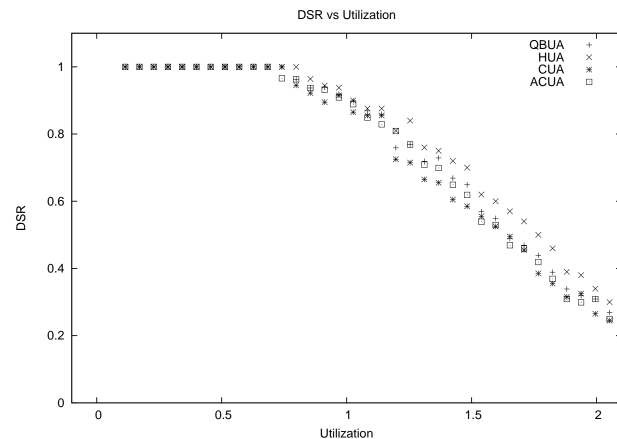

Fig. 12. TMR vs. Utilization (no failures), thread set III

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.

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. Ford, B., Lepreau, J.: Evolving mach 3.0 to a migrating thread model. In: USENIX Technical Conference. (1994) 97–114
4. OMG: Real-time corba 2.0: Dynamic scheduling specification. Technical report, Object Management Group (2001)
5. Jensen, E., Locke, C., Tokuda, H.: A time driven scheduling model for real-time operating systems (1985) IEEE RTSS, pages 112–122, 1985.

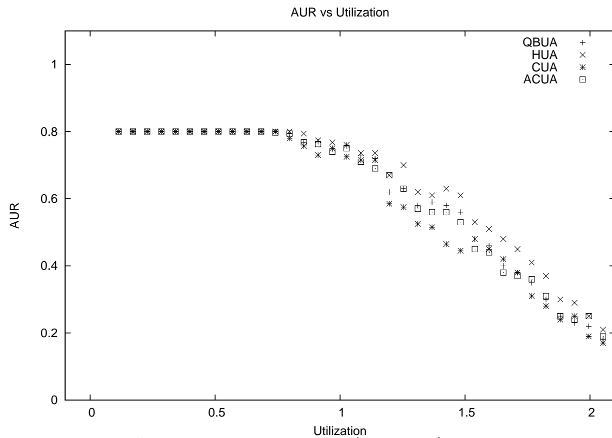

Fig. 13. AUR vs. Utilization (failures), thread set III

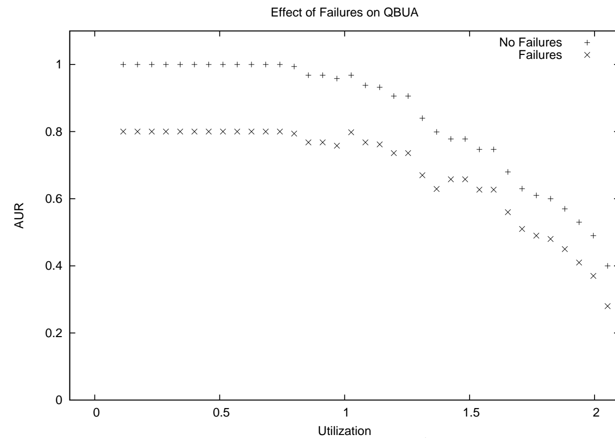

Fig. 14. Effect of failures of QBUA, thread set III

6. Locke, C.D.: Best-Effort Decision Making for Real-Time Scheduling. PhD thesis, CMU (1986) CMU-CS-86-134.
7. Clark, R.K.: Scheduling Dependent Real-Time Activities. PhD thesis, CMU (1990) CMU-CS-90-155.
8. Curley, E., Anderson, J.S., Ravindran, B., Jensen, E.D.: Recovering from distributable thread failures with assured timeliness in real-time distributed systems. In: IEEE SRDS. (2006) 267–276
9. Ravindran, B., Anderson, J.S., Jensen, E.D.: On distributed real-time scheduling in networked embedded systems in the presence of crash failures. In: IFIP SEUS Workshop. (2007) 67–81
10. Ravindran, B., Curley, E., Anderson, J.S., Jensen, E.D.: On best-effort real-time assurances for recovering from distributable thread failures in distributed real-time systems. In: ISORC '07, IEEE Computer Society (2007) 344–353
11. Goldberg, J., Greenberg, I., et al.: Adaptive fault-resistant systems (chapter 5: Adpative distributed thread integrity). Technical Report csl-95-02, SRI International (1995)
12. Hermant, J.F., Lann, G.L.: Fast asynchronous uniform consensus in real-time distributed systems. IEEE Transactions on Computers **51**(8) (2002) 931 – 944
13. 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
14. Hermant, J.F., Widder, J.: Implementing reliable distributed real-time systems with the Θ-model. In: OPODIS. (2005) 334–350
15. 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`.
16. Chen, W., Toueg, S., Aguilera, M.K.: On the quality of service of failure detectors. IEEE Transactions on Computers **51**(1) (2002) 13–32
17. Maynard, D.P., Shipman, S.E., et al.: An example real-time command, control, and battle management application for alpha. Technical Report Archons Project 88121, CMU CS Dept. (1988)
18. Sterzbach, B.: GPS-based clock synchronization in a mobile, distributed real-time system. Real-Time Syst. **12**(1) (1997) 63–75
19. Halang, W.A., Wannemacher, M.: High accuracy concurrent event processing in hard real-time systems. Real-Time Syst. **12**(1) (1997) 77–94
20. Dana, P.H.: Global positioning system (gps) time dissemination for real-time applications. Real-Time Syst. **12**(1) (1997) 9–40
21. Druschel, P., Rowstron, A.: PAST: A large-scale, persistent peer-to-peer storage utility. In: HOTOS '01. (2001) 75–80
22. 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
23. McCanne, S., Floyd, S.: (ns-2: Network Simulator) http://www.isi.edu/nsnam/ns/.