Scheduling Distributable Real-Time Threads in the Presence of Crash Failures and Message Losses

Sherif F. Fahmy
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
fahmy@vt.edu

Binoy Ravindran
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
binoy@vt.edu

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

ABSTRACT

We consider the problem of scheduling distributable real-time threads under run-time uncertainties including those on thread execution times, thread arrivals, node failures, and message losses. We present a distributed scheduling algorithm called ACUA that is designed under a partially synchronous model, allowing for probabilistically-described message delays. We show that ACUA satisfies thread time constraints in the presence of crash failures and message losses, is early-deciding, and has an efficient message and time complexity. The algorithm has also better "best-effort" real-time property than past thread scheduling algorithms.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

Keywords

Scheduling, Distributed Systems, Utility Accrual.

1. INTRODUCTION

Some emerging networked embedded systems are dynamic in the sense that they operate in environments with dynamically uncertain properties (e.g., [2]). These uncertainties include transient and sustained 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. Another 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 [9], and later in the Real-Time CORBA 1.2 standard directly provides such a model as their first-class programming and scheduling abstraction. A distributable thread is a single thread of execution with a globally unique identity that transparently extends and retracts through local and remote objects. We focus on distributable threads as our end-to-end programming/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., [9–11]) can be broadly categorized into two classes: *independent node scheduling* and *collaborative scheduling*. In the independent scheduling approach (e.g., [9, 11]), threads are scheduled at nodes using propagated thread scheduling parameters and without any interaction with other nodes. Fault-management is separately addressed by *thread integrity protocols* 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 [9]. In the collaborative scheduling approach (e.g., [10]), nodes explicitly cooperate to construct system-wide thread schedules, anticipating and detecting node failures using FDs.

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, this results in systems with low coverage—i.e., the high likelihood for the resulting timing assurances to be violated, when the synchrony assumptions are violated at run-time (e.g., due to overloads, or other exigencies). 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 [1], Aguilera *et. al.* describe the design of a fast FD for synchronous systems and show how it can be used to solve the consensus problem for real-time systems. Their algorithm achieves the optimal bound for both message and time complexities. In [6], Hermant and Widder describe the *Theta*-model, where only the ratio, Θ, between the fastest and slowest message is known. This increases the coverage of algorithms designed under

SAC'08, March 16-20, 2008, Fortaleza, Ceará, Brazil.

this model, as less assumptions are made about the system. Though Θ is sufficient for proving the correctness of such algorithms, an upper bound on communication delay is needed to establish timeliness properties.

In this paper, we target partially synchronous systems, and consider the partially synchronous model in [4], where message delay and message loss are probabilistically described. For such a model, we design a collaborative thread scheduling algorithm called the *Asynchronous Consensus-based Utility accrual scheduling Algorithm* (or ACUA). We show that ACUA satisfies thread time constraints in the presence of crash failures and message losses, is early-deciding (i.e., its decision time is proportional to the actual number of crashes), and has a message and time complexity that compares favorably with other algorithms in its class. Furthermore, we show that ACUA 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 [10,11]. We also prove the exception handling properties of ACUA. To the best of our knowledge, this is the first collaborative thread scheduling algorithm designed under a partially synchronous model.

The rest of the paper is organized as follows: In Section 2, we discuss our models and objectives. Section 3 discusses the rationale of ACUA, and Section 4 describes the algorithm. We establish ACUA's properties in Section 5, report experimental results in Section 6, and conclude in Section 7.

2. MODELS AND OBJECTIVE

2.1 Models

Distributable Threads. Distributable threads execute in local and remote objects by location-independent invocations and returns. The portion of a thread executing an object operation is called a *thread segment*. Thus, a thread can be viewed as being composed of a concatenation of thread segments. A thread can also be viewed as being composed of a sequence of *sections*, where a section is a maximal length sequence of contiguous thread segments on a node.

We assume that execution time estimates of the 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 $\mathbf{T} = \{T_1, T_2, \ldots\}$. 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) [7]. A TUF allows us to decouple the urgency of a thread from its importance. This decoupling is a key property allowed by TUFs since the urgency of a thread may be orthogonal to its importance. A thread T_i's TUF is denoted as $U_i(t)$. A classical deadline is unit-valued—i.e., $U_i(t) = \{0, 1\}$, since importance is not considered. Downward step TUFs generalize classical deadlines where $U_i(t) = \{0, \{m\}\}$. We focus on downward step TUFs, and denote the maximum, constant utility of a TUF $U_i(t)$, simply as U_i. Each TUF has an initial time I_i, which is the earliest time for which the TUF is defined, and a termination time X_i, which, for a downward step TUF, is its discontinuity point. $U_i(t) > 0, \forall t \in [I_i, X_i]$ and $U_i(t) = 0, \forall t \notin [I_i, X_i], \forall i$.

System Model. We consider a set of nodes $\Pi = \{1, \cdots, N\}$, with a logical communication channel between each pair of nodes. 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 [9]. The dual processor assumption is used to reduce ACUA's scheduling overhead. (In Section 5, we show that with such a scheduling co-processor, a thread need only survive ACUA's overhead once). We assume that communication links are unreliable, i.e., messages can be lost with probability p, and communication delay is described by some probability distribution.

Exceptions and Abort Model. Each section of a thread has an associated exception handler. We consider a termination model for thread failures including termination time violations and node failures. When such thread failures occur, the section exception handlers are triggered to restore the system to a safe state. The exception handlers may have time constraints expressed as TUFs. A handler's TUF's initial time is the time of failure of the handler's thread. The handler's TUF's termination time is relative to its initial time. Thus, a handler's absolute and relative termination times are *not* the same. Each handler also has an execution time estimate. This estimate along with the handler's TUF are described by the handler's thread when the thread arrives at a node. A handler is marked as ready for execution when either its latest start time (see Section 3.2) expires, or it receives an explicit invocation from its successor.

Failure Model. The nodes are subject to crash failures. Up to $f_{max} \leq n - 1$ nodes may fail. The actual number of failures in the system is denoted as $f \leq f_{max}$.

2.2 Scheduling Objectives

Our primary objective is to design a thread scheduling algorithm that will maximize the total utility accrued by all threads as much as possible. 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. ALGORITHM RATIONALE

ACUA 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 ACUA 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.

In ACUA, when a thread arrives into the system, each node suggests a set of threads for rejection from the system based on its local environment. The nodes must then agree on a set of threads to reject from the system-wide schedule. We formulate this problem as a consensus problem with the following properties: (a) If a correct node decides on a reject set $rSet$, then some node proposed $rSet$; (b) Nodes do not decide on different reject sets (*Uniform agreement*); (c) Every correct node eventually decides (i.e. termination).

Since consensus is part of ACUA, we need to solve it on the partially synchronous model we consider in this work. For consensus to be solvable on a given system model, it must support the implementation of one of the Chandra-Toueg unreliable failure detectors [3]. In Section 3.1, we show that it is possible to design an S class FD [3] using the QoS FD described in [4], and thereby show that consensus is solvable

in our system. Specifically, we show that it is possible to design a FD that provides the semantics of an S class FD *with very high probability* for the *duration of the consensus algorithm.* Past work [10,11] had considered the existence of a perfect FD (P class FD), since they considered a fully synchronous model. In this work, we use the S class FD (which is weaker than a P class FD) because we consider partially synchronous systems. An S class FD has the following properties: 1) Completeness Property:- There is a time, T_D, after which a failed node is permanently suspected by all nodes; and 2) Accuracy Property:- There is some correct node that is never suspected by all other nodes.

In addition, since the time constraint of a thread is specified using a TUF, we need to decompose this thread-wide TUF into section TUFs in order to allow each node to perform its local scheduling. We describe this in Section 3.2.

3.1 Failure Detection

We use the QoS FD in [4] to implement our S class FD. This QoS FD 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, $suspect_i$. Our consensus algorithm polls this aggregate FD every δ time unit when its service is required.

From [4] we know that the probability, P_A, that the result of one of the QoS FDs is accurate when it is queried at random is $E(T_G)/E(T_{MR})$, where $E(T_G)$ is the average time that the FD's output remains correct and $E(T_{MR})$ is the average time between consecutive mistakes. We also know that $E(T_G) = E(T_{MR}) - E(T_M)$, where $E(T_M)$ is the average time it takes for the FD to correct an erroneous failure suspicion. Both $E(T_{MR})$ and $E(T_M)$ are input QoS values chosen when designing the FD, thus we can control P_A by choosing appropriate values for these two parameters.

To show that we can implement an S-class FD using the QoS FD in [4], we need to determine when the consensus algorithm needs the service of the FD. The consensus algorithm used in ACUA is the quorum-based algorithm in [8] which requires the service of the FD in line 5 only.

In the worst case, the algorithm takes N rounds (in each of the first $N-1$ rounds an erroneous suspicion of the round coordinator leads to the next round until round N is reached). Let Δ be the communication delay described by the probability density function $delay(t)$ and the cumulative distribution function $DELAY(t)$, the consensus algorithm will spend either Δ to receive the coordinator's estimate or T_D to detect the coordinator's failure (T_D is an input QoS parameter chosen when designing the FD, and since it is the time after which a failed node will be permanently suspected, it satisfies the completeness property of an S class FD).

In the worst case, the consensus algorithm will query the FD n times, where $n = \frac{N \times T_D}{\delta}$. We consider each of these queries to be an independent experiment with probability $p = 1 - P_A$ of resulting in an erroneous suspicion. Therefore, the probability that the FD monitoring a single node makes at least one erroneous suspicion during the execution of the algorithm is $P_{FDM} = 1 - bino(0, n, p)$, where $bino(x, n, p)$ is the binomial distribution with parameters n and p. Since there are $N-1$ FDs on each node, the probability that a given node erroneously suspects x nodes is given by $bino(x, N-1, P_{FDM})$ and the probability that a node suspects a majority of the nodes in the system is $\sum_{i=\frac{N-1}{2}+1}^{N-1} bino(i, N-1, P_{FDM})$. Using this analysis, we constructed a FD that suspected a majority of nodes with probability 1.5×10^{-110}. We believe this probability is too low to be of practical concern for the time scales we consider. Therefore, since it is not practically possible for a node to erroneously suspect a majority of other nodes during the execution of the consensus algorithm, the set of nodes not suspected by all nodes in the system have to intersect in at least one node. That node is never suspected by any of the other nodes in the system, thus satisfying the accuracy property of an S class FD. In addition, the T_D detection time of our FD satisfies the completeness property of an S class FD. Therefore, we are able to implement an S class FD with very high probability on our system during the execution of our consensus algorithm.

3.2 TUF Decomposition

Thread time constraints are expressed using TUFs. 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 of a thread, we derive its termination time as the thread's termination time. 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:

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

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 D above, is a random variable Δ, as mentioned in Section 3.1. Therefore, the value of D 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. This is further explored in Section 5.

As mentioned in Section 2.1, each handler has a TUF that specifies its **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 which allows us to delay the execution of the exception handlers as much as possible, thus leaving room for more important threads. Therefore, in the equations below we replace t_f with $S_k^i.tt$, the termination time of thread i's last section:

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

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, 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 decom-

position, we compute start times for each handler:

$$S_j^h.st = \{\ S_j^h.tt - S_j^h.ex \quad 1 \leq i \leq k$$

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

4. ALGORITHM DESCRIPTION

Algorithm 1 shows the general structure of ACUA. Algorithm 1 is triggered when a thread arrives into the system or when a node fails When ACUA is triggered, each node constructs a local schedule (line 5). In lines 6-14 each node suggests a set of threads for rejection based on the local schedule it constructs in line 5. In line 15, the nodes send the set of threads they suggest for rejection to all other nodes in the system. Each node then waits for a certain time period to collect the suggestions that other nodes send (lines 15-16). Using these suggestions, each node makes a decision about which set of threads should be rejected from the system (line 18). A consensus protocol is then started in order to reach agreement among the nodes about the set of threads that will be rejected, using the decision each node made in line 18 as input to the consensus protocol (line 19). After reaching agreement, the nodes remove the set of rejected threads from their waiting queue (line 20) and construct a new local schedule containing the remaining threads (line 21).

A important part of ACUA is how it selects a set of threads for rejection locally (lines 7-14). ACUA distinguishes between threads that become unschedulable due to local overloads, and threads that become unschedulable in order to accommodate a newly arrived thread. This is necessary because a newly arrived thread can only be accepted into the system if *all* its future head nodes accept its sections. Thus, if some nodes reject other threads' sections in order to accommodate the arriving thread, and other nodes reject the sections of the arriving thread, the new thread should not be accepted into the system and the sections rejected to accommodate the new thread's sections on some nodes should be allowed to execute normally.

Lines 7-12 perform this function. If the section of the newly arrived thread is not part of the constructed schedule, it cannot be responsible for the elimination of other threads from the system. Thus the difference between the current schedule and the previous schedule is the set of threads that the node proposes for rejection (lines 8-9). On the other hand, if a section of the newly arrived thread is part of the schedule, we need to differentiate between two possible causes for rejecting threads: 1) overload conditions may render some threads unschedulable and 2) the newly arrived thread may render some threads unschedulable.

The former set can be determined by constructing a schedule without considering S_j^i (line 7) and then subtracting that set from the set of previously schedulable threads (line 11). On line 12 we place a separator, $\perp$, between the set of threads rendered unschedulable due to overload and the set of threads rendered unschedulable due to the acceptance of a section of the newly arrived thread. Note that nodes indicate whether they accept, reject, or are not responsible for the sections of a newly arrived thread by prepending 1, 0 and $\emptyset$ to their suggestions respectively.

Using this additional information, the problem mentioned above can be eliminated by only eliminating threads rendered unschedulable by an arriving thread if ***all*** its future head nodes accept the thread. The details of this functionality is contained in the function *DetRejectSet*. Note that the timeout value on line 16 is a stochastic value, thus even if none of the nodes fail, there is a non-zero probability that some nodes do not receive the suggestions of all other nodes. This is further addressed in Section 5.

Algorithm 1: ACUA on each node i

```
1:  input: σ_r^i; // σ_r^i: unordered ready queue;
2:  input: σ_p; // σ_p : previous schedule;
3:  output σ_i; // σ_i: schedule;
4:  Initialization: Σ_i = ∅; w_i = ∅;
5:  σ_i = ConstructSchedule(σ_r^i);
6:  if i is head node for newly arrived thread j then
7:      σ_tmp = ConstructSchedule(σ_r^i − S_j^i);
8:      if S_j^i ∉ σ_i then
9:          rSet = 0 ∪ (σ_p − σ_i);
10:     else
11:         tmp = (σ_p − σ_tmp);
12:         rSet = 1 ∪ (σ_p − (σ_i − S_j^i) − tmp) ∪ ⊥ ∪ tmp;
13: else
14:     rSet = ∅ ∪ (σ_p − σ_i);
15: send(rSet_i, i, t) to all;
16: upon receive(rSet_j, j) until 2D do
17:     Σ_i = Σ_i ∪ rSet_j;
18: w_i=DetRejectSet(Σ_i);
19: w_i=UniformConsensus(w_i);
20: UpdateSectionSet(w_i, σ_r^i);
21: σ_i =ConstructSchedule(σ_r^i);
22: σ_p = σ_i;
23: return σ_i;
```

Algorithm 2: DetRejectSet on node i

```
1:  input: Σ_i; // Σ_i: set of suggestions for rejection.
2:  output w_i; // w_i: rejection set output.
3:  accept=true;
4:  w_i = ∅;
5:  for each future head node, j, of newly arrived thread do
6:      tmp_j=retrieve node j's entry from Σ_i;
7:      if head(tmp_j)=0 then
8:          accept=false;
9:  for each node j do
10:     rSet_j=retrieve node j's entry from Σ_i;
11:     rSet_j = rSet_j - first element in rSet_j;
12:     if j is a future head node then
13:         if accept=true then
14:             w_i=w_i ∪ elements before and after ⊥ in rSet_j;
15:         else
16:             w_i=w_i ∪ only elements after ⊥ in rSet_j;
17:     else
18:         w_i=w_i ∪ rSet_j;
19:     if node j is a head node for thread set Γ with a
        section on node i then
20:         if node i does not receive node j's suggestion then
21:             w_i=w_i ∪ Γ;
22: return w_i;
```

Algorithm 2 describes how nodes determine the set of threads to suggest for rejection from the system. The algorithm first checks whether the newly arrived thread has been accepted into the system by all future head nodes (lines 5-8). Lines 10 and 11 retrieve the suggestion of node j and remove the first element. Lines 12-16 determine which threads to consider for rejection based on the fact that threads ren-

dered unschedulable by the newly arrived thread on some nodes should only be rejected if ***all*** head nodes accept the sections of the newly arrived thread. Line 18 adds the set of threads that non-head nodes suggest for rejection. Finally, lines 19-21 suggests threads for rejection if they have a section hosted on the current node and the current node does not receive any suggestions from one of the previous, current, or future head nodes of the threads. This is done because a node suspects those nodes it does not receive suggestions from to have failed, and thus suggests for elimination the threads that are hosted by them. The uniform consensus algorithm we use is described in [8].

We now turn our attention to the scheduling algorithm that nodes use to construct a local schedule. This algorithm is encapsulated by the function *ConstructSchedule* (see Algorithm 3). The algorithm takes a list of sections, and constructs a total order with each section's global Potential Utility Density (or PUD). The global PUD of a section is the ratio of the utility of the thread that the section belongs to, to the sum of the remaining execution times of all the thread's sections. The algorithm examines each section in the PUD-order, including them in the schedule, and testing for schedule feasibility. If infeasible, the inserted section is rejected, and the process is repeated until all sections are examined. Note that we construct a total order on *global* Potential Utility Density (PUD) in order to attempt to maximize system-wide accrued utility. This can be seen in line 8 of the algorithm, where the execution time of the whole thread, $T_i.ex$, is used instead of the execution time for each individual section, $S_i.ex$, when computing PUD. The algorithm for *UpdateSectionSet* is simple and involves a simple removal of the rejected threads from a node's ready queue.

Algorithm 3: ConstructSchedule

```
1:  input: σ_r,σ_p,H; output σ;
2:  Initialization: t = t_cur;σ = ∅;HandlerIsMissed = fasle;
3:  for each S_i ∈ σ_p such that S_i ∉ σ_r do
4:      Insert(S_j^h, H, S_j^h.tt);
5:  σ = H;
6:  for each S_i ∈ σ_r do
7:      if S_{j-1}^i.tt + D + S_j^i.ex ≤ S_j^i.tt then
8:          S_i.PUD = min( U_i(t+T_i.ex)/T_i.ex , U_i^h(t+T_i.ex+T_i^h.ex)/(T_i.ex+T_i^h.ex) )
9:      else
10:         S_i.PUD = 0
11: σ_tmp=sortByPUD(σ_r);
12: for each S_i ∈ σ_tmp from head to tail do
13:     if S_i.PUD ≥ 0 then
14:         Insert(S_i, σ, S_i.tt);
15:         Insert(S_i^h, σ, S_i^h.tt);
16:         if Feasible(σ) = false then
17:             Remove(S_i, σ, S_i.tt);
18:             if S_i^h ∉ H then
19:                 Remove(S_i^h, σ, S_i^h.tt);
20:     else
21:         break;
22: σ_p = σ;
23: return σ;
```

5. ALGORITHM PROPERTIES

We compare the best-effort properties of ACUA, CUA [10], and HUA [11]. In HUA (an *independent* node scheduling algorithm), thread sections are scheduled locally at each node they arrive at using their propagated scheduling parameters. The local scheduler is a modified version of DASA [5], which uses the heuristic of favoring tasks with a high utility to execution time ratio, i.e., high PUD, when constructing the schedule. These modifications allow HUA to manage the scheduling of exception handlers in case of thread failure.

In CUA (a *collaborative* scheduling algorithm), when a thread arrives, its sections are sent to all its future head nodes. Each node constructs its schedule locally according to a modified version of DASA. The nodes then cooperate with each other to reach agreement on a system-wide set of threads eligible for execution. Basically, this agreement step involves the elimination of any threads that have any of their sections missing from the global schedule.

We quantify the best-effort property by introducing the concept of DASA Best Effort (or DBE) property:

DEFINITION 1. *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.*

LEMMA 1. *HUA, does not have the DBE property.*

PROOF. The proof is by counterexample. Assume that a system has two nodes, n_1 and n_2, and two threads, T_1 and T_2. Assume that each thread has two sections, one hosted on each of the nodes. Let the sections be S_1^1 and S_1^2 for T_1 and S_2^1 and S_2^2 for T_2. Assume that both threads have end-to-end step-down TUFs, with the utility for T_1 being 5 and the utility of T_2 being 6. Also assume that both threads arrive at n_1 at t_0. Assume that the execution times of S_1^1, S_1^2, S_2^1 and S_2^2 are 2, 3, 3 and 1 time units respectively and that both threads have a relative termination time of 5.

The parameters above ensure that only one of the threads can be scheduled successfully. Therefore, an algorithm that has the DBE property would choose T_2 for execution since its global PUD, $\frac{6}{4} = 1.5$, is greater than the PUD of T_1, $\frac{5}{5} = 1$. Note that the DBE property will result in a system-wide accrued utility of 6 in this case.

In contrast, HUA computes the PUD of the *sections* of each thread hosted on each node when constructing its schedule [11]. Since the PUD of S_1^1, $\frac{5}{2} = 2.5$, is greater than the PUD of S_2^1, $\frac{6}{3} = 2$, the scheduler on n_1 will choose S_1^1 for scheduling first. By the time S_1^1 has finished execution, $t_0 + 3$, releasing S_2^1 for execution will mean that it will finish past the global termination time of T_2 ($T_0 + 5$). Thus, only T_1 will execute with a resulting accrued utility of 5 for the system.

Thus HUA does not have the DBE property. □

LEMMA 2. *CUA does not have the DBE property.*

PROOF. CUA does not have the DBE property became it does not schedule all feasible threads in the system. For example, if two nodes host sections of two threads, T_1 and T_2, during overloads, one node may schedule the section belonging to T_2 at the expense of that belonging to T_1 and the other may schedule the section belonging to T_1 at the expense of that belonging to T_2. Since CUA excludes threads from the system if they are missing any of their sections and both of the above threads have one of their sections missing, both threads will be excluded from the system. This is unnecessary since excluding one thread will render the other

schedulable, thus the algorithm does not schedule all feasible threads and therefore does not have the DBE property. □

THEOREM 3. *ACUA has the DBE property for threads that can be delayed $O(f\Delta + nk)$ (see Lemma 5) and are still schedulable.*

PROOF. ACUA overcomes the issue mentioned in Theorem 1 because it uses the PUD of the entire thread when constructing local schedules on each node. Thus sections that are excluded are those with the least *system-wide* PUD. In other words, the threads in ACUA are considered in non-increasing order of global PUD for scheduling. In addition, ACUA overcomes the issue mentioned in Theorem 2 by preventing an arriving thread from eliminating other threads if at least one of the nodes that will be hosting a future head of the arriving thread does not accept that section for scheduling. The details of this procedure are explained in Algorithms 1 and 2. This allows ACUA to schedule all feasible threads. Thus all feasible threads that can tolerant the scheduling overhead of ACUA and still remain feasible will be scheduled in non-increasing order of global PUD. The theorem follows from Definition 1. □

THEOREM 4. *ACUA can tolerate up to $f_{max} = n - 1$ faulty processors.*

PROOF. This follows directly from the fault tolerant property of the S class based consensus algorithm in [8] which we use in our work. □

LEMMA 5. *ACUA has time complexity $O(f\Delta + nk)$.*

PROOF. Lines 5 and 7 in Algorithm 1 have complexity $O(k^2)$ where k is the maximum number of sections in the ready queue of system nodes. Lines 8-15 have constant complexity, lines 16-17 have complexity 2Δ, line 18 has complexity $O(nk)$, line 19 has complexity $O((f+1)\Delta)$, line 20 has complexity $O(k)$ and line 21 has complexity $O(k^2)$. Therefore, the algorithm has actual complexity of $3k^2 + 2\Delta + nk + (f+1)\Delta + k$, which is asymptotically $O(f\Delta + nk)$ if we consider k a constant. □

This time complexity compares favorably with the time complexity of CUA, which is $O(D + df + nk)$ [10], asymptotically. However, the value of the time complexity of CUA is lower than that of ACUA since it makes the additional assumption of the existence of a fast FD [1]. In addition, Δ is a random variable, thus the timing guarantee for ACUA is stochastic in nature.

LEMMA 6. *ACUA has message complexity $O(fn^2)$.*

PROOF. Lines 16-17 in Algorithm 1 have message complexity n (one for each suggested rejection set sent by a node). Line 19 has message complexity $n^2(f+1)$ since each round has a message cost of n^2. The algorithm is early deciding so it will take $f+1$ rounds [8]. Therefore the actual message cost of the algorithm is $n + n^2(f+1)$, which is asymptotically $O(fn^2)$. □

The message complexity of ACUA is asymptotically higher than that of CUA, $O(nf)$ [10]. However we show in our next theorem that the *size* of each message should be smaller for ACUA in well behaved systems.

LEMMA 7. *The message size in ACUA is smaller than that in CUA for well behaved systems.*

PROOF. The input to the consensus algorithm in ACUA is the set of rejected threads while the input to the consensus algorithm in CUA is the set of schedulable threads. Since the set of rejected threads should be smaller than the set of accepted threads in well behaved systems, we claim that the message size in ACUA is smaller than that in CUA. □

LEMMA 8. *If each section of a thread meets its derived termination time (see Section 3.2), then under ACUA, 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 in Section 3.2 are violated during runtime.

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

LEMMA 9. *If all nodes are underloaded and no nodes fail, then no threads will be suggested for rejection by ACUA with high, computable, probability p_{norej}.*

PROOF. Since the nodes are all underloaded and no nodes fail, Algorithm 3 ensures that all sections will be accepted. Thus, the only source of thread rejection is if a node does not receive a suggestion from other nodes during the timeout value, D, (see Algorithm 1 in Section 4). This can occur due to one of two reasons; 1) the broadcast message (line 15), that indicates the start of the consensus algorithm, may not reach some nodes 2) the broadcast message reaches all nodes, but these nodes do not send their suggestions to other nodes in the system 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 $p = 1 - DELAY(D)$. 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 of consensus 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 this message is received, a node waits for messages from all other nodes. The probability that none of these messages arrive after the timeout is $tmp = bino(0, N, p)$. Since there are N nodes, the probability that none of these nodes miss a message is $bino(N, N, tmp)$. Therefore 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 receive suggestions from all other nodes in response to this start of consensus message i.e. $p_{norej} = bino(N, N, tmp) \times P_{tmp}$. □

THEOREM 10. *If all nodes are underloaded, no nodes fail (i.e. $f = 0$), and threads can be delayed $O(f\Delta + nk)$ time units once and still be schedulable, ACUA meets all the thread termination times yielding optimal total utility with high, computable, probability, P_{alg}.*

PROOF. By Lemma 9, 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 3. Thus, by Lemma 8, each thread, j, will meet its termination time with probability p^j_{suc}. Therefore, for a system with X 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}, $P_{alg} = P_{tmp} \times p_{norej}$.

ACUA takes $O(f\Delta + nk)$ time units to determine a newly arrived thread's schedulability. If this delay causes any of the thread's sections to miss their termination times, the thread will not be schedulable. We require that a thread suffer this delay *once* because we assume that there is a scheduling co-processor 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 11. *ACUA is an early deciding algorithm that achieves consensus on the system-wide execution eligible thread set in a partially synchronous system with virtually certain probability.*

PROOF. Since the consensus algorithm in [8], on which we base our algorithm, is early deciding so is our algorithm. In addition, we show in Section 3.1 that we can provide an S class FD with very high probability during the execution of our algorithm (with probability of error 1.50×10^{-110}), therefore the S class FD based consensus algorithm in [8] executes on our system with virtually certain probability. Since the input to the consensus algorithm is the set of threads to reject from the system, at its completion all nodes will agree on the set of threads to reject from their schedules and hence on the system-wide set of execution eligible threads. □

THEOREM 12. *If $n - f$ nodes do not crash, are underloaded, and all incoming threads can be delayed $O(f\Delta + nk)$ and still be schedulable, ACUA meets the execution time of all threads in its eligible execution thread set, Γ, with high computable probability, P_{alg}.*

PROOF. By Theorem 11, ACUA achieves system-wide consensus on the set of schedulable threads. By Lemma 9, the probability that none of the threads hosted by the surviving nodes are rejected is, $p_{norej} = bino(N-f, N-f, tmp) \times tmp$ where $tmp = bino(0, N-f, p)$ and $p = 1 - DELAY(D)$. Thus all sections belonging to those threads will be scheduled to meet their derived termination times. By Lemma 8, this implies that each of these threads, j, will meet its termination time with probability p^j_{suc}. Therefore, for a system with an eligible thread set, Γ, the probability that all threads meet their termination times is $P_{tmp} = \prod_{j \in \Gamma} p^j_{suc}$. Thus, the probability that all the remaining threads are accepted is $P_{alg} = P_{tmp} \times p_{norej}$. □

DEFINITION 2 (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 13. *If a node hosting a section, S^i_j, of thread T_i fails at time t_f, every correct node will include handlers for thread T_i in H by time $t_f + T_D + t_a$, where t_a is an implementation-specific computed execution bound for ACUA calculated per the analysis in Theorem 5.*

PROOF. Since the QoS FD we use detects a failed node in T_D time units [4], all nodes detect the failure of the failed node at time $t_f + T_D$. As a result, ACUA is triggered and excludes T_i from the system because nodes will not receive any suggestions from node j (see lines 19-21 of Algorithm 2). Consequently, Algorithm 3 will include the section handlers for this thread in H (see lines 3-4 of Algorithm 3). Execution of ACUA completes in time t_a and thus all handlers will be included in H by time $t_f + T_D + t_a$. □

LEMMA 14. *If a section S_i, where $i \neq k$, fails at time t_f (per Definition 2) and section S_{i+1} is correct, then under ACUA, its handler S^h_i will be released no earlier than S^h_{i+1}'s completion and no later than $S^h_{i+1}.tt + D + S^h_i.X - S^h_i.ex$.*

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

For the first case, we know from the analysis in Section 3.2 that the start time of S^h_i is $S^h_{i+1}.tt + S^h_j.X + D - S^h_j.ex$. Thus, by definition, it satisfies the upper bound in the theorem. Also, since $S^h_j.X \geq S^h_j.ex$ (otherwise the handler would not be schedulable), $S^h_{i+1}.tt + S^h_j.X + D - S^h_j.ex > S^h_{i+1}.tt$, and this satisfies the lower bound of the theorem.

For the second case, an explicit message has arrived indicating the completion of S^h_{i+1}. Since the message was sent, this means that $S^h_{i+1}.tt$ has already passed, thus satisfying the theorem lower bound. Further, the message should have arrived D time units after S^h_{i+1} finishes execution (i.e., at $S^h_{i+1}.tt + D$), since $S^h_{i+1}.tt + D \leq S^h_{i+1}.tt + D + S^h_i.X - S^h_i.ex$ (as $S^h_i.X \geq S^h_i.ex$), thus satisfying the upper bound. □

LEMMA 15. *If a section S_i fails (per Definition 2), then under ACUA, its handler S^h_i will complete no later than $S^h_i.tt$ (barring S^h_i'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 means that S_i and its handler were previously determined to be feasible before $S_i.tt$ and $S^h_i.tt$, respectively (lines 13-19, Algorithm 3). When some previous head node of S_i's thread fails, ACUA will be triggered and will remove S_i from the pending queue. In addition, Algorithm 3 will include S^h_i in H and construct a feasible schedule containing S^h_i (lines 3-21). Since the schedule is feasible and S^h_i is inserted to meet $S^h_i.tt$ (line 4), then S^h_i will complete by time $S^h_i.tt$ □

THEOREM 16. *When a thread fails, 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 $S^h_i.X$, which fails at time t_f, and (distributed) scheduler latency t_a, this bound is $T_i.X + \sum_i S^h_i.X + kD + T_D + t_a$.*

PROOF. The LIFO property follows from Lemma 14. Since it is guaranteed that each handler, S^h_i, cannot begin before the termination time of handler S^h_{i+1} (the lower bound in Lemma 14), thus we guarantee LIFO execution of the handlers. Lemma 15 shows that all correct handlers complete in bounded 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 13) and ACUA guarantees that all these sections will complete before their termination times (Lemma 15). 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 3.2) is $T_i.X + \sum_i S_i^h.X + kD + T_D + t_a$. The theorem follows. □

6. EXPERIMENTAL RESULTS

We performed a series of simulation experiments to compare the performance of ACUA to 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.

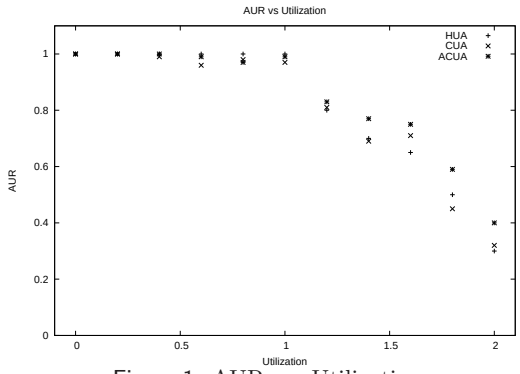

Figure 1: AUR vs. Utilization

Figure 1 shows the result of the AUR experiments. As can be seen, during underloads (i.e., for utilizations less than one), all three algorithms perform near optimally. However, both ACUA and CUA sometimes accrue less than 100% AUR during underloads. This is due to the higher overhead of these algorithms. It is during overloads, however, that ACUA begins to outperform both of the other algorithms. The performance of CUA and HUA varied during overloads; sometimes CUA had better performance and at other times the situation was reversed. This difference depends on which loss of best-effort property, Lemma 1 or Lemma 2, causes the greatest loss of utility for the thread set being tested. Thus, it can be seen that ACUA has a better best-effort property than CUA and HUA, validating Lemmas 1 and 2 and Theorem 3. In Figure 2, we show the TMR of the algorithms, which exhibit the same trend.

7. CONCLUSIONS

We presented a best-effort utility accrual scheduling algorithm, ACUA, for scheduling distributable real-time threads in partially synchronous systems. We compared ACUA in terms of its best-effort property, and message and time complexity to two previous thread scheduling algorithms including CUA and HUA. We showed that ACUA has a better best-effort property during overloads than HUA and CUA, and has message and time complexities that are comparable to CUA (which is in its class). We also showed the exception handling properties of ACUA.

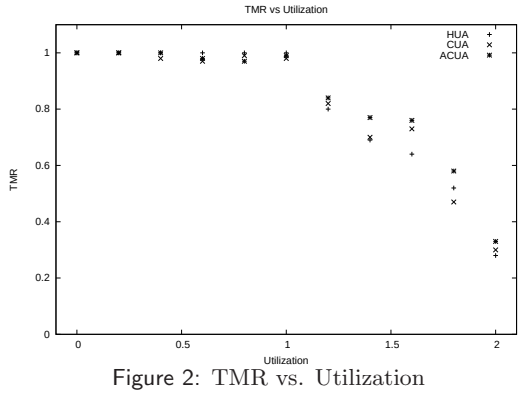

Figure 2: TMR vs. Utilization

8. REFERENCES

[1] M. K. Aguilera et al. On the impact of fast failure detectors on real-time fault-tolerant systems. In *DISC*, pages 354–370. Springer-Verlag, 2002.

[2] J. R. Cares. *Distributed Networked Operations: The Foundations of Network Centric Warfare*. iUniverse, Inc., 2006.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[4] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, 2002.

[5] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, CMU, 1990. CMU-CS-90-155.

[6] J.-F. Hermant and J. Widder. Implementing reliable distributed real-time systems with the Θ-model. In *OPODIS*, pages 334–350, 2005.

[7] E. Jensen, C. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems. In *IEEE RTSS*, pages 112–122, 1985.

[8] A. Mostéfaoui and M. Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. In *DISC*, pages 49–63, 1999.

[9] J. D. Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems — The Alpha Kernel*. Academic Press, 1987.

[10] B. Ravindran, J. S. Anderson, and E. D. Jensen. On distributed real-time scheduling in networked embedded systems in the presence of crash failures. In *IFIP SEUS Workshop*, 2007.

[11] B. Ravindran, E. Curley, et al. On best-effort real-time assurances for recovering from distributable thread failures in distributed real-time systems. In *IEEE ISORC*, pages 344–353, 2007.