On Distributed Real-Time Scheduling in Networked Embedded Systems in the Presence of Crash Failures

Binoy Ravindran[1], Jonathan S. Anderson[1], and E. Douglas Jensen[2]

[1] ECE Dept., Virginia Tech, Blacksburg, VA 24061, USA, andersoj@vt.edu, binoy@vt.edu
[2] The MITRE Corporation, Bedford, MA 01730, USA, jensen@mitre.org

Abstract. We consider the problem of scheduling distributable real-time threads in networked embedded systems that operate under run-time uncertainties including those on thread execution times, thread arrivals, and node failure occurrences. We present a distributed scheduling algorithm called CUA. We show that CUA satisfies thread time constraints in the presence of crash failures, is early-deciding, has an efficient message complexity of $O(fn)$ (where f is the number of crashes that actually occur and n is the number of nodes), and is time-optimal with a time lower bound of $O(D+fd+nk)$ (where D is the message delay upper bound, d is the failure detection bound, and k is the maximum number of threads). In crash-free runs, the algorithm constructs schedules within $O(D+nk)$, and yields optimal total utility if nodes are also not overloaded. The algorithm is also "best-effort" in that a high importance thread that may arrive at any time has a very high likelihood for feasible completion (in contrast to classical admission control algorithms which favor feasible completion of admitted threads over admitting new ones, irrespective of thread importance).

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.

Many 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, and arbitrary node failures. 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 contention 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), MK7.3 [4], Real-Time CORBA 1.2 [5], and Sun's emerging Distributed Real-Time Specification for Java (DRTSJ) [6] directly provide 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.

When overloads occur, meeting time constraints of all threads is impossible as the demand exceeds the supply. The urgency of a thread is sometimes orthogonal to the relative importance of the thread—-e.g., the most urgent thread may be the least important, and vice versa; the most urgent may be the most important, and vice versa. Hence when overloads occur, completing the most important threads irrespective of thread urgency is desirable. Thus, a distinction has to be made between urgency and importance during overloads. (During underloads, such a distinction generally need not be made, especially if all time constraints are deadlines, as optimal algorithms exist that can meet all deadlines—e.g., EDF [7].)

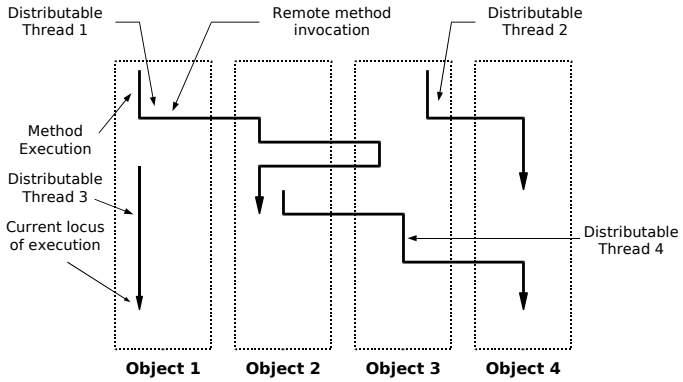

Fig. 1. Four Distributable Threads

Deadlines cannot express both urgency and importance. Thus, we consider the *time/utility function* (or TUF) timeliness model [8] 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.

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 (see [9] for example algorithms).

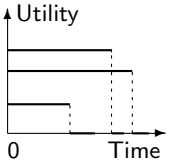

Fig. 2. Example Step TUF Time Constraints

UA algorithms that maximize total utility under downward step TUFs (e.g., [10, 11]) 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" [10] 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: Assured Thread Timeliness in the Presence of Failures. In this paper, we consider the problem of scheduling threads in the presence of the previously mentioned uncertainties, focusing particularly on (arbitrary) node failures. Past efforts on thread scheduling (e.g., [2, 12, 13]) consider a paradigm broadly called *independent node scheduling*, where 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* [14] that run concurrent to thread execution. Thread integrity protocols detect failures of the thread abstraction, delivering failure-exception notifications [2, 13]. This approach avoids the overhead of inter-node communication, and is therefore message-efficient and tractable (solely from the thread

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

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.

We consider instead *collaborative scheduling*, where nodes explicitly cooperate to construct system-wide thread schedules, anticipating node failures. Of course, doing so incurs message overhead costs, and thus raises fundamental questions including a) what upper bounds can be established for such message costs, and b) what are the consequent payoffs.

We answer these questions. We present an algorithm called *Consensus-based Utility accrual scheduling Algorithm* (or CUA). The algorithm considers distributable threads that are subject to TUF time constraints. Threads may have arbitrary arrival behaviors, may exhibit unbounded execution time behaviors (causing node overloads), and may span nodes that are subject to arbitrary crash failures. For such a model, we consider the scheduling objective of maximizing the total thread accrued utility.

CUA is a distributed algorithm that consists of a set of node schedulers that cooperate to realize the algorithm's logic. Node schedulers invoke themselves at events of interest (e.g., thread arrival, failure-suspicion), construct local schedules, broadcast schedules, and arrive at consensus on system-wide schedules, despite failures. We show that CUA satisfies thread time constraints in the presence of crash failures, is early-deciding (i.e., its decision time is proportional to the actual number of crashes), has an efficient message complexity of $O(fn)$ (where f is the number of crashes that actually occur and n is the number of nodes), and is time-optimal with a time lower bound of $O(D+fd+nk)$ (where D is the message delay upper bound, d is the failure detection bound, and k is the maximum number of threads). Note that early-deciding consensus algorithms in the continuous-time synchronous model (where processes do not execute in lock-step rounds) have an optimal time lower bound of $O(D + fd)$ [15]. In crash-free runs, CUA constructs schedules within $O(D + nk)$, and yields optimal total utility if nodes are also not overloaded. The algorithm also retains the fundamental best-effort property of UA algorithms—i.e., a high importance thread that may arrive at any time has a very high likelihood for feasible completion. To the best of our knowledge, this is the first algorithm to provide these properties.

The rest of the paper is organized as follows: In Section 2, we discuss the models of our work and state the algorithm objectives. Section 3 presents CUA. We establish the algorithm's properties in Section 4. We conclude the paper in Section 5.

2 Models

2.1 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. Further details of the thread model can be found in [2, 5, 16].

Execution time estimates of the sections of a thread are assumed to be known when the thread arrives. A section's execution time estimate is the execution time estimate of the contiguous set of thread segments that starts from the operation of the object invoked on the node (i.e., the first thread segment executed on the node) and ends with the first remote invocation made from the node. The time estimate includes that of the section's normal code and its exception handler 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 $\mathbf{T} = \{T_1, T_2, T_3, \ldots\}$. The set of sections of a thread T_i is denoted as $[S_1^i, S_2^i, \ldots, S_k^i]$.

2.2 Timeliness Model

We specify the time constraint of each thread using a TUF. 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 (e.g., Figure 2), 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 $U_i(t) = 0, \forall t \notin [I_i, X_i], \forall i$.

2.3 System and Failure Models

Our system and failure models follow that of [15]. We consider a system model where a set of processing components, generically referred to as *nodes*, denoted by the totally-ordered set $\Pi = \{1, 2, \ldots, n\}$, are interconnected via a network. We consider a single hop network model (e.g., a LAN), with nodes interconnected through a hub or a switch. The system is assumed to be (partially) synchronous in that there exists an upper bound D on the message delivery latency. A reliable message transmission protocol is assumed; thus messages are not lost or duplicated. Node clocks are assumed to be perfectly synchronized, for simplicity in presentation. The CUA algorithm, however, can be extended to clocks that are nearly synchronized with bounded drift rates.

As many as f_{max} nodes may crash arbitrarily. The actual number of node crashes is denoted as $f \leq f_{max}$. Nodes that do not crash are called *correct*.

Each node is assumed to be equipped with a perfect failure detector [18] that provides a list of nodes that are deemed to have crashed. If a node q belongs to such a list of node p, then node p is said to *suspect* node q. The failure detection time [19] $d \leq D$ is assumed to be bounded. Similar to [15], for simplicity in presentation, we assume that D is a multiple of d. Failure detectors are assumed to be (a) *accurate*—i.e., a node suspects a node q only if q has previously crashed; and (b) *timely*—i.e., if a node q crashes at time t, then every correct node permanently suspects q within $t + d$.

2.4 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 of UA algorithms (described in Section 1).

3 The CUA Algorithm

3.1 Rationale and Design

In the absence of crash failures, there is no compelling motivation for nodes to collaborate for constructing a system-wide schedule.[4] Thus, we consider crash failures and first establish the premise for collaboration—i.e., why thread scheduling in the presence of crash failures should consider node collaboration. We first define the notion of a thread's *current head node* and *future head nodes*:

Definition 1 (Current Head Node). *The current head node of a thread T_i is the node where T_i is currently executing (i.e., where T_i's head is currently located).*

Definition 2 (Future Head Nodes). *The future head nodes of a thread T_i are those nodes where T_i will make remote invocations in the future.*

The crash of a node p affects other nodes in the system in three possible ways: (a) p may be the current head node of one or more threads; (b) p may be the future head node of one or more threads; and (c) p may be the current and future head node of one or more threads.

[4] One motivation for such a collaboration would be to construct an optimized system-wide schedule — one that can result in greater timeliness (e.g, total accrued utility) than what would be possible without collaboration. We do not consider such a node collaboration as that is outside the scope of this work.

If p is only the current head node of one or more threads, then all nodes in the system which are future head nodes of those threads are immediately affected, since they can now release the processor time for scheduling those future heads and use it for scheduling other threads. If p is only the future head node of one or more threads, all nodes in the system which are (also) future head nodes of those threads are affected, since they can now similarly release the processor time for scheduling other threads. There may be a set of nodes which are not future head nodes of p's threads. Only those nodes are unaffected.

This implies that when a node p crashes, a system-wide decision must be made (by all those nodes which are affected by p's crash) regarding which set of threads are eligible for execution in the system—referred to as an *execution-eligible thread set*—and which are not. Furthermore, this decision must be made in the presence of failures, since nodes may crash while that decision is being made. We formulate this problem as a *consensus* problem [20] with the following properties: (a) If a correct node decides an eligible thread set $\mathcal{T}$, then some correct node proposed $\mathcal{T}$; (b) Nodes (correct or not) do not decide different execution-eligible sets (*uniform agreement*); (c) Every correct node eventually decides (i.e., termination).

Observe that the first property is stronger than the uniform validity property of the (uniform) consensus problem specification. Uniform validity states that if a node decides a value, then some node previously proposed that value. For the thread scheduling problem, this would mean that it would be possible for correct nodes to decide on an execution-eligible thread set that was previously proposed by a node, which later crashed. Consequently, this will result in an invalid system-wide execution-eligible thread set. Thus, we qualify the uniform validity property with *correct*.

Now that a premise for node collaboration is established, we need to determine how a node can propose a set of threads that should be eligible for execution. Since the task model is dynamic—i.e., when threads will be created is entirely arbitrary and statically unknown, future scheduling events cannot be considered at a scheduling event.[5] Thus, the execution-eligible thread set must be constructed solely exploiting the current system knowledge. Since the primary scheduling objective is to maximize the total thread accrued utility, and it may not be possible to meet all thread termination times due to overloads, a reasonable heuristic for determining the execution-eligible thread set is a "greedy" strategy: Favor "high return" threads over low return ones, and complete as many of them as possible before thread termination times.

The potential utility that can be accrued by executing a thread section on a node defines a measure of that section's "return on investment." We measure this using a metric called the *Potential Utility Density* (or PUD). On a node, a thread section's PUD measures the utility that can be accrued per unit time by immediately executing the section on the node.

Thus, each node (that is a current head node for one or more threads) examines thread sections in its local ready queue for potential inclusion in a feasible schedule for the node in the order of decreasing section PUDs. For each section, the algorithm examines whether that section can be completed early enough, allowing successive sections of the thread to also be completed early enough, to allow the entire thread to meet its termination time. We call this property, the feasibility of a section. If the section is infeasible (due to schedule overload), it is rejected. The process is repeated until all sections are examined, yielding a local schedule of feasible sections.

To determine section feasibility, we assign termination times for each section of a thread (derived from the thread's termination time) in a way that allows the thread's termination time to be met if each of the section termination times are met. The termination time that a section must meet to allow the thread to meet its termination time is simply the thread termination time if the section is the last section; otherwise, it is the latest start time of the section's successor section minus the communication delay upper bound. The latest start time of a section is the section's termination time minus its estimated execution time. Thus, the section termination times of a thread T_i with k sections are given by:

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

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.

Thus, the local schedule constructed by a node p is an ordered list of a subset of sections in p's ready queue that can be feasibly completed, and will likely result in high local accrued utility (due to the greedy nature of the PUD heuristic). The set of threads, say T_p, of these sections included in p's schedule is

[5] A "scheduling event" is an event that invokes the scheduling algorithm.

proposed by p as those that are eligible for system-wide execution, from p's standpoint. However, not all threads in T_p may be eligible for system-wide execution, because the current and/or future head nodes of some of those threads may crash. Consequently, the set of threads that are eligible for system-wide execution is that subset of threads with no absent sections from their respective current and/or future head node schedules.

3.2 Algorithm Description

The CUA algorithm that we present is derived from Aguilera *et. al*'s time-optimal, early-deciding, uniform consensus algorithm in [15]. A pseudo-code description of CUA on each node i is shown in Algorithm 1.

Algorithm 1: CUA: Code for each node i

```
1: input: σ_r^i; output: σ_i; // σ_r^i: unordered ready queue of node i's sections; σ_i: schedule
2: Initialization: Σ_i = ∅; ω_i = ∅; max_i = 0;
3: σ_i = ConstructLocalSchedule(σ_r^i);
4: send(σ_i, i) to all;
5: upon receive (σ_j, j) until 2D do // After time 2D, consensus begins
6:    Σ_i = Σ_i ∪ σ_j;
7: ω_i = DetermineSystemWideFeasibleThreadSet(Σ_i);
8: upon receive (ω_j, j) do
9:    if j > max_i then max_i = j; ω_i = ω_j;
10: at time (i − 1)d do
11:    if suspect j for any j : 1 ≤ j ≤ i − 1 then
12:       ω_i = DetermineSystemWideFeasibleThreadSet(Σ_i \ σ_j);
13:       send(ω_i, i) to all;
14: at time (j − 1)d + D for every j : 1 ≤ j ≤ n do
15:    if trust j then decide ω_i;
16: UpdateSectionSet(ω_i, σ_r^i);
17: σ_i = ConstructLocalSchedule(σ_r^i);
18: return σ_i;
```

The algorithm is invoked at a node i at the scheduling events including 1) creation of a thread at node i and 2) inclusion of a node k into node i's suspect list by i's failure detector.

When invoked, a node i first constructs the local section schedule by invoking `ConstructLocalSchedule()` (line 3). This procedure accepts i's (unordered) local ready queue of sections σ_r^i and returns a schedule (an ordered list) σ_i. The node then sends this schedule (σ_i, i) to all nodes (line 4). This message contains a header that indicates that consensus will start within $2D$ time units of the sender's message transmission—i.e., one D for the sender's message and one D for the recipients to respond (line 5). Recipients are expected to immediately respond by constructing their local section schedules and sending them to all nodes. When node i receives a schedule (σ_j, j), it includes that schedule into a schedule set Σ_i (line 6). Thus, after $2D$ time units, all nodes have a schedule set containing all schedules received.

A node i then determines its consensus decision (i.e., system-wide execution-eligible thread set) by calling procedure `DetermineSystemWideFeasibleThreadSet()`. This procedure accepts i's schedule set Σ_i and determines that subset of threads with no absent sections from their respective (current head and/or future head) node schedules in Σ_i. Node i uses a variable ω_i to maintain its consensus decision. Node i now starts the consensus process.

The algorithm divides real-time in consecutive rounds of duration d each, where node i's round (or round i) corresponds to the time interval $[(i-1)d, id)$. At the beginning of round i, node i checks whether it suspects *any* of the nodes with a smaller node ID. If so, it sends (ω_i, i) to all nodes (line 11). Note that for $i = 1$, node i will send $(\sigma_1, 1)$ to all nodes (if i does not crash), since no nodes have an ID lower than 1. Also, note that the messages sent in a round could be received in a higher round since $D > d$.

Each node i maintains a variable max_i that contains the ID of the largest-ID node from which it has received a consensus proposal (max_i is initialized to zero). When a node i receives a proposed execution-eligible thread set (ω_j, j) that is sent from another node j with an ID that is larger than max_i (i.e., $j > max_i$), then i updates its consensus decision to thread set ω_j and max_i to j (line 9).

At times $(j-1)d + D$ for $j = 1, \ldots, n$, node i is guaranteed to have received potential consensus proposals from node j. Thus, at these times, i checks whether j has crashed; if not, i arrives at its consensus decision on the thread set ω_i (line 15).[6]

Node i then updates its ready queue σ_r^i by removing those sections whose threads are absent in the consensus decision ω_i. The updated ready queue is used to construct a new local schedule σ_i, which is returned by the algorithm. The head section of this schedule is subsequently dispatched for execution.

3.3 Constructing Section Schedules

We now describe the algorithm `ConstructLocalSchedule()`. To describe this algorithm, we first define a few auxiliary functions. Since this algorithm is not a distributed algorithm per se, we drop the suffix i from notations σ_r^i (input unordered list) and σ_i (output schedule), and refer to them as σ_r and σ, respectively. Similarly, sections are referred to as S_i, for $i = 1, 2, \ldots$, except when reference to their distributable threads is needed.

- `sortByPUD(`σ`)` returns a schedule ordered by non-increasing section PUDs. If two or more sections have the same PUD, then the section(s) with the largest execution time estimate will appear before any others with the same PUD.
- `Insert(`S_i`,`σ`,`I`)` inserts section S_i in the ordered list σ at the position indicated by index I; if entries in σ exists with the index I, S_i is inserted before them. After insertion, S_i's index in σ is I.
- `Remove(`S_i`,`σ`,`I`)` removes section S_i from ordered list σ at the position indicated by index I; if S_i is not present at the position in σ, the function takes no action.

Algorithm 2: Algorithm `ConstructLocalSchedule()`

```
1:  input: σ_r; output: σ;
2:  Initialization: t := t_cur, σ := ∅;
3:  for each section S_i ∈ σ_r do
4:      if S^i_{j-1}.tt + D + S^i_j.ex > S^i_j.tt then
            // If j = 1, then S^i_{j-1}.tt = 0
5:          abort(S_i);
        else
6:          S_i.PUD = U_i(t + S_i.ex)/S_i.ex;
7:  σ_tmp := sortByPUD(σ_r);
8:  for each section S_i ∈ σ_tmp from head to tail do
9:      if S_i.PUD > 0 then
10:         Insert(S_i, σ, S_i.tt);
11:         if ScheduleFeasible(σ)=false then
12:             Remove(S_i, σ, S_i.tt);
13:     else break;
14: return σ;
```

Algorithm 2 describes the local section scheduling algorithm. When invoked at time t_{cur}, the algorithm first checks the feasibility of the sections. If the earliest predicted completion time of a section is later than its termination time, it can be aborted (line 4). Otherwise, the algorithm calculates the section's PUD (line 6).

The sections are then sorted by their PUDs. In each step of the *for*-loop from line 8 to 13, the section with the largest PUD is inserted into σ, if it can produce a positive PUD. The schedule σ is maintained in the non-decreasing order of section termination times. Thus, a section S_i is inserted into σ at a position that corresponds to S_i's termination time ($S_i.tt$) in σ's non-decreasing termination time order.

After inserting a section S_i, the schedule σ is tested for feasibility (line 11; Algorithm 3). If σ becomes infeasible, S_i is removed. After examining all sections, the ordered list σ is returned.

[6] If node i receives a proposed execution-eligible thread set (ω_j, j) from another node j at times $(i-1)d$ or $(j-1)d + D$, we assume that the node executes line 8 before it executes line 10 or line 14 (similar to [15]).

Algorithm 3: Algorithm `ScheduleFeasible()`

```
1: input: σ; output: true or false;
2: Initialization: CumExecTime = 0;
3: for each section S^i_j ∈ σ do
4:     CumExecTime = CumExecTime + S^i_{j-1}.tt + D + S^i_j.ex; // If j = 1, then S^i_{j-1}.tt = 0
5:     if CumExecTime > S^i_j.tt then return false;
6: return true;
```

Algorithm 3 determines the feasibility of a schedule σ. A schedule σ is feasible if the predicted completion time of each section S_i in σ, denoted $S_i.ct$, does not exceed S_i's termination time $S_i.tt$ (line 5). $S_i.ct$ is the time at which S_i is released on its node plus the sum of the execution times of all sections that occur before S_i in σ and S_i's execution time $S_i.ex$. Note that except for current thread head nodes, Algorithm 1 is invoked before sections are actually released on future thread head nodes. Thus, we calculate a section S_i's release time as the termination time of S_i's predecessor (i.e., $S^i_{j-1}.tt$) plus the message delay upper bound D, since that is the latest time by which S_i must be released on its node.

Algorithm 2 therefore seeks to include those sections in the schedule that are likely to result in high total utility (due to the greedy nature of the PUD heuristic). Further, since the invariant of schedule feasibility is preserved throughout the examination of sections, the output schedule is always a feasible schedule. Thus, during underloads, schedule σ will always be feasible in line 11 (Algorithm 2), the algorithm will never reject a section, and will produce a schedule which is the same as that produced by EDF (where deadlines are equal to section termination times). Consequently, this schedule will meet all section termination times during underloads.

During overloads, the algorithm will reject one or more sections to construct a feasible schedule. Due to the algorithm's greedy nature, the rejected sections are less likely to contribute a total utility that is larger than that contributed by the accepted sections.

Asymptotic Complexity. The cost of Algorithm 2 is dominated by the *for*-loop (line 8 to 13) which iterates at most k times for a ready queue with k sections. The cost of this loop is dominated by Algorithm 3, which costs $O(k)$ to test the feasibility of a schedule with k sections. Thus, Algorithm 2's asymptotic cost is $O(k^2)$.

4 Algorithm Properties

We first describe CUA's timeliness property under crash-free runs:

Theorem 1. *If all nodes are underloaded and no nodes crash (i.e., $f_{max} = 0$), CUA meets all thread termination times, yielding optimum total utility.*

Proof. From the discussion in Section 3.3, if a node is underloaded, Algorithm 2 will meet all section termination times at the node. Thus, if all nodes are underloaded and $f_{max} = 0$, all section termination times are met. If all sections of a thread meet their termination times, then the thread will meet its termination time by virtue of Equation 1. Theorem follows.

Theorem 2. *CUA achieves (uniform) consensus (i.e., uniform validity, uniform agreement, termination) on the system-wide execution-eligible thread set in the presence of up to f_{max} failures.*

Proof. This is self-evident from the algorithm description and follows from [15].

Theorem 3. *CUA's time complexity is $O(D + fd + nk)$ and message complexity is $O(fn)$.*

Proof. If the maximum number of sections at a node is k, then `ConstructLocalSchedule` costs $O(k^2)$. Procedure `DetermineSystemWideFeasibleThreadSet` will cost $O(nk)$ to examine at most n schedules sent by n nodes, with each schedule containing at most k sections. Thus, lines 3-7 of Algorithm 1 has an actual time cost of $2D + \delta_1$, where δ_1 measures the actual cost of $O(k^2) + O(nk)$. These steps will involve n messages, one for each schedule sent by a node in line 4.

Lines 8–15 has an actual time cost of $D + fd$ and will involve $(f + 1)n$ messages [15]. Line 12, executed at most f times adds computational cost $O(fnk)$, and `UpdateSectionSet` will remove at most k sections

in its schedule, costing $O(k)$, and `ConstructLocalSchedule` costs $O(k^2)$, resulting in a combined actual cost of a constant, say δ_2.

Thus, Algorithm 1 has an actual time cost of $2D + \delta_1 + D + fd + \delta_2$, or $3D + \delta + fd$, and will involve $n + (f+1)n$, or $(f+2)n$ messages. The corresponding asymptotic costs are $O(D + fd + nk)$ and $O(fn)$, respectively (for $n \geq k$ and $f \geq 2$). When $f = f_{max}$, the algorithm thus constructs schedules in at least $3D + \delta + f_{max}d$ time, or $O(D + f_{max}(d + nk))$. When $f_{max} = 0$ (i.e. crash-free), the algorithm constructs schedules in time $3D + \delta$, or $O(D + nk)$.

From Theorems 2 and 3, we obtain the algorithm's early-deciding property:

Theorem 4. *CUA is a time-optimal, early-deciding algorithm that achieves consensus on the system-wide execution-eligible thread set.*

Proof. From Theorem 3, CUA decides in time proportional to f. From Theorem 2, the algorithm achieves consensus on the system-wide execution-eligible thread set. From [15], no early-deciding algorithm (in the continuous-time synchronous model, where processes do not execute in lock-step rounds) has a time bound lower than $D + fd$. Theorem follows.

We now establish CUA's timeliness property in the presence of failures.

Theorem 5. *If $n - f$ nodes (i.e., correct nodes) are underloaded, then CUA meets the termination times of all threads in its (consensus decision of) execution-eligible thread set.*

Proof. From Theorem 4, $n - f$ nodes arrive at the same decision on the system-wide execution-eligible thread set, say $\mathcal{T}$. If these nodes are under-loaded, then CUA meets the termination times of all threads in $\mathcal{T}$, per Theorem 1.

To establish the algorithm's best-effort property (Section 1), we first define the concept of a *Non Best-effort time Interval* (or NBI):

Definition 3. *Consider a distributable thread scheduling algorithm $\mathcal{A}$. Let a thread T_i be created at a node at a time t with the following properties: (a) T_i and all threads in $\mathcal{A}$'s execution-eligible thread set at time t are not feasible (system-wide) at t, but T_i is feasible just by itself; and (b) T_i has the highest PUD among all threads in $\mathcal{A}$'s execution-eligible thread set at time t. Now, $\mathcal{A}$'s NBI, denoted $NBI_{\mathcal{A}}$, is defined as the duration of time that T_i will have to wait after t, before it is included in $\mathcal{A}$'s execution-eligible thread set. Thus, T_i is assumed to be feasible at $t + NBI_{\mathcal{A}}$.*

We now describe the NBI of CUA and other distributable thread scheduling UA algorithms including DASA [11], LBESA [10], and AUA [13] under crash-free runs. Note that DASA, LBESA, and AUA are thread scheduling algorithms that belong to the independent node scheduling paradigm (i.e., they make their scheduling decisions using propagated thread scheduling parameters and without collaborating with other nodes). Since we focus on crash-free runs, the presence of a thread integrity protocol that these algorithms use for thread fault-management can be ignored.

Theorem 6. *Under crash-free runs (i.e., $f_{max} = 0$), the worst-case NBI of CUA is $3D + \delta$, DASA's and LBESA's is δ, and that of AUA is $+\infty$.*

Proof. CUA will examine T_i at t, since the arrival of a new thread is a scheduling event. Since T_i has the highest PUD and is feasible system-wide, the algorithm will arrive at a consensus decision on an execution-eligible thread set that includes T_i in time $3D + \delta$ when $f_{max} = 0$, per Theorems 2 and 3.

DASA and LBESA will examine T_i at t (at the node where T_i was created), since a thread arrival is also a scheduling event for them. Further, since T_i has the highest PUD and is feasible, they will include T_i's first section in their feasible (local) schedules at t, yielding a worst-case NBI of δ, the time constant involved for the algorithm to arrive at the local decision. This cost δ will be the same as that of CUA, since DASA's and LBESA's asymptotic computational costs are the same as that of CUA (i.e., $O(k^2)$).

AUA will examine T_i at t, since a thread arrival at any time is also a scheduling event under it. However, AUA is a TUF/UA algorithm in the classical admission control mould (e.g., [21]) and will reject T_i in favor of previously admitted threads, yielding a worst-case NBI of $+\infty$.

5 Conclusions and Future Work

We presented a distributed real-time scheduling algorithm called CUA. The algorithm considers distributable threads with TUF time constraints, arbitrary thread arrival behaviors, thread execution overrun behaviors causing overloads, and arbitrary crash failures. We showed that CUA satisfies thread time constraints in the presence of crash failures, is early-deciding, has an efficient message complexity of $O(fn)$, and is time-optimal with a time lower bound of $O(D+fd+nk)$. In crash-free runs, the algorithm constructs schedules within $O(D+nk)$, and yields optimal total utility if nodes are also not overloaded. We also showed that the algorithm has a tightly bounded non-best-effort time interval, which implies that a high importance thread that may arrive at any time has a very high likelihood for feasible completion.

Our work just scratched the surface of a very rich problem space, and so many directions exist for immediate and long-term study. Example directions include considering asynchronous models (e.g., [22, 23]), allowing synchronization dependencies between threads (e.g., due to mutually exclusive sharing of non-processor resources), considering ad hoc network infrastructures (e.g., mobile, wireless networks), and developing non-deterministic (e.g., probabilistic) timing assurances.

References

1. CCRP: Network centric warfare. `http://www.dodccrp.org/ncwPages/ncwPage.html`
2. Northcutt, J.D.: Mechanisms for Reliable Distributed Real-Time Operating Systems — The Alpha Kernel. Academic Press (1987)
3. Ford, B., Lepreau, J.: Evolving Mach 3.0 to a migrating thread model. In: USENIX Technical Conference. (1994) 97–114
4. The Open Group: MK7.3a Release Notes. The Open Group Research Institute, Cambridge, Massachusetts (October 1998)
5. OMG: Real-time CORBA 2.0: Dynamic scheduling specification. Technical report, Object Management Group (September 2001)
6. Jensen, E.D., Wellings, A., Clark, R., Wells, D.: The distributed real-time specification for Java: A status report. In: Proceedings of The Embedded Systems Conference. (2002)
7. Horn, W.: Some simple scheduling algorithms. Naval Research Logistics Quaterly **21** (1974) 177–185
8. Jensen, E.D., et al.: A time-driven scheduling model for real-time systems. In: IEEE RTSS. (Dec. 1985) 112–122
9. Ravindran, B., Jensen, E.D., Li, P.: On recent advances in time/utility function real-time scheduling and resource management. In: IEEE ISORC. (May 2005) 55 – 60
10. Locke, C.D.: Best-Effort Decision Making for Real-Time Scheduling. PhD thesis, CMU (1986)
11. Clark, R.K.: Scheduling Dependent Real-Time Activities. PhD thesis, CMU (1990)
12. Kao, B., Garcia-Molina, H.: Deadline assignment in a distributed soft real-time system. IEEE TPDS **8**(12) (Dec. 1997) 1268–1274
13. 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
14. Goldberg, J., Greenberg, I., et al.: Adaptive fault-resistant systems (chapter 5: Adpative distributed thread integrity). Technical Report csl-95-02, SRI International (January 1995) `http://www.csl.sri.com/papers/sri-csl-95-02/`.
15. Aguilera, M.K., Lann, G.L., Toueg, S.: On the impact of fast failure detectors on real-time fault-tolerant systems. In: DISC '02: Proceedings of the 16th International Conference on Distributed Computing, London, UK, Springer-Verlag (2002) 354–370
16. Anderson, J., Jensen, E.D.: The distributed real-time specification for Java: Status report. In: JTRES. (2006)
17. Maynard, D.P., Shipman, S.E., et al.: An example real-time command, control, and battle management application for Alpha. Technical Report Archons Technical Report 88121, CMU CS Dept. (December 1988)
18. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2) (1996) 225–267
19. Chen, W., Toueg, S., Aguilera, M.K.: On the quality of service of failure detectors. IEEE Transactions on Computers **51**(5) (May 2002) 561–580
20. Lynch, N.: Distributed Algorithms. Morgan Kaufmann (1996)
21. Bestavros, A., Nagy, S.: Admission control and overload management for real-time databases. In: Real-Time Database Systems: Issues and Applications. Kluwer Academic Publishers (1997)
22. Fetzer, C., Schmid, U., Susskraut, M.: On the possibility of consensus in asynchronous systems with finite average response times. In: ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05), Washington, DC, USA, IEEE Computer Society (2005) 271–280
23. Hermant, J.F., Widder, J.: Implementing reliable distributed real-time systems with the *theta*-model. In: OPODIS. (2005) 334–350