

Design and Implementation of a Consensus-Driven, Lazy-Abort Distributable Thread Scheduling Algorithm

Jonathan S. Anderson and Binoy Ravindran

*Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg Virginia, 24061, USA
{andersonj, binoy}@vt.edu*

E. Douglas Jensen

*The MITRE Corporation
Bedford, Massachusetts, 01730, USA
jensen@mitre.org*

Abstract

We demonstrate an improved consensus-driven utility accrual scheduling algorithm (DUA-CLA) for distributable threads which execute under run-time uncertainties in execution time, arrival models, and node crash failures. The DUA-CLA algorithm's message complexity ($O(fn)$), lower time complexity bound ($O(D + fd + nk)$), and failure-free execution time ($O(D + nk)$) are established, where D is the worst-case communication delay, d is the failure detection bound, n is the number of nodes, and f is the number of failures. DUA-CLA is shown to have the "lazy-abort" property — abortion of currently-infeasible tasks is deferred until there is no possibility of completing the task on time. Further, it exhibits "schedule-safety" — segments (and therefore, threads) proposed as feasible for execution by a node which fails during the consensus decision will be removed from the consensus set and will not cause an otherwise-feasible segment to be excluded. These properties mark improvements over earlier strategies in common- and worst-case performance. Finally, we present our implementation of DUA-CLA in a DRTSJ-compliant Java virtual machine, and quantitative results are presented validating fundamental properties of the algorithm.

1 Introduction

1.1 Dynamic Distributed Real-Time Systems

Distributed real-time systems, such as those found in industrial automation, net-centric warfare (NCW), and military surveillance, require support for timely, end-to-end activities. Timeliness includes application-specific acceptability of end-to-end time constraint satisfaction, and of the predictability of that satisfaction. These activities may include computational, sensor, and actuator steps which levy a causal (and thus sequential) ordering of operations, contingent on interactions with physical systems. Such end-to-end tasks may be represented in a concrete distributed system as: chains of (a) nested remote method invocations; (b) publish, receive steps in a publish-subscribe framework; (c) event occurrence and event handlers.

Dynamic Systems. The class of distributed real-time systems under consideration here, typified by NCW applications [1], is characterized by dynamically uncertain execution properties due to transient and persistent local overloads, uncertain task arrival patterns and rates, uncertain communication delays, node failures (which, contrary to usual assumptions, may be clustered in space and time), and variable resource demands. However, while local activities in these systems may have timeliness requirements with sub-second magnitudes, end-to-end tasks commonly have considerably larger magnitudes of milliseconds to multiple minutes.

Consequently it is possible — and necessary — to reason non-deterministically (for example, probabilistically) about end-to-end activity timeliness. Despite the larger timeliness magnitudes, these activities are mission-critical and thus require the strongest assurances possible under the circumstances.

End-to-End Context. In order for the distributed system to make appropriate resource allocation decisions in keeping with the end-to-end activity requirements (e.g., timeliness, security) expressed by the designers and users, some representation of these parameters and the current state of the end-to-end activity must be provided. The *distributable thread* abstraction, itself a generalization of the familiar operating system thread, provides just such an extensible model for reasoning about end-to-end activity behavior.

Distributable threads (hereafter, simply *threads*) appeared first in the Alpha OS [2] and were adopted in Mach 3.0 [3] and Mk7.3 [4]. Recently, this abstraction has served as the basis for RT-CORBA 2.0 [5] and Sun’s emerging Distributed Real-Time Specification for Java (DRTSJ) [6], where threads form the primary programming abstraction for concurrent, distributed activities. In RT-CORBA and DRTSJ, threads represent a single, system-wide concurrent locus of execution, with globally unique identity available for use in synchronization and resource contention.

Time Constraints for Overloaded Systems. In underloaded systems it is sufficient to provide an assessment of the *urgency* of an activity, typically in the form of a *deadline*. For these scenarios, known-optimal algorithms (e.g., EDF [7]) exist to meet all deadlines, given certain restrictions on the system and its load. When a system is overloaded, it is by definition unable to meet the timeliness requirements of all activities. As a consequence, resource managers must decide *which* subset of activities to complete, and with *what degree of timeliness*.

This requires the system must be aware of the relative *importance* of each activity. In general, the importance of an activity may be orthogonal to its urgency; therefore, deadlines are insufficient to make acceptable resource allocation decisions. Furthermore, the importance — or *utility* — of completing an activity may vary depending on the time the activity is completed.

We consider therefore the *time/utility function* (or TUF) timeliness model [8], in which the utility of completing an activity is given as a function of its completion time. For simplicity, in this paper we confine our work to downward step-shaped TUFs, in which an activity’s utility is a constant U_i when the task is completed before a deadline t ; if the task is completed after the deadline it has zero utility. Traditional deadlines are a special case where all $U_i = 1$.

Utility Accrual Scheduling. When time constraints are expressed as TUFs, it becomes natural to express resource allocation (e.g., scheduling and shared resource contention resolution) decisions in terms of *utility accrual* (UA) based criteria. A common UA criteria is *maximize summed utility*, in which resource allocation decisions are made such that the total summed utility accrued is maximized. Several such UA scheduling and sequencing heuristics have been investigated (e.g., [9, 10]).

UA scheduling algorithms that maximize total accrued utility for activities described by downward-step TUFs default to EDF during underload conditions, achieving optimum schedules. During overloads, these algorithms favor higher-utility activities over those with lower-utility. The resulting “best-effort” adaptive behavior exhibits graceful degradation as load increases, shedding the lowest-utility work irrespective of its urgency.

Admission control is not viable as a form of overload management because it does not consider that newly arriving activities may have greater importance than that of previously admitted activities. In dynamic systems, admission cannot necessarily constitute a contract to complete an activity at an acceptable time or even at all.

1.2 Contributions and Related Work

The central contributions of this paper are: (a) the Distributed Utility Accrual - Consensus-based Lazy Abort (DUA-CLA) scheduling algorithm; (b) theoretical bounds on DUA-CLA’s timeliness, message efficiency, and optimality behavior in a variety of conditions; (c) a practical implementation of DUA-CLA in the DRTSJ middleware; and (d) experimental results illustrating the validity of the theoretical results.

This paper presents significant new progress on recent work published by the authors in [11]. DUA-CLA expands the theoretical performance envelope of our earlier work in two significant ways: First, we have introduced the “lazy-abort” property (see Theorem 7), which relaxes the aggressive task-abortion behavior present in our earlier work while maintaining asymptotic execution times and performance assurances. Second, DUA-CLA is shown (see Theorem 8) to be “schedule-safe” in the presence of failures during distributed rescheduling. This property means that such failures will not cause overly pessimistic feasibility assessments, resulting in unnecessary rejection of tasks rendered feasible by partial failures in other tasks.

We distinguish the theoretical and experimental results presented in this paper from our recent and ongoing related work on gossip-based approaches [12] in the models and objectives, as well as algorithm assurances. While [12] focuses on ad hoc network infrastructures with message omission failures and probabilistic communication delays, providing probabilistic timing assurances, this paper addresses fixed network infrastructures with deterministic message delays, providing stronger, deterministic timing assurances. Furthermore we consider the collaborative scheduling approach with the goal of understanding the effectiveness and concomitant trade offs of that approach rather than the independent scheduling strategy described in the gossip-based approach.

The DUA-CLA algorithm represents a unique approach to distributable thread scheduling in two respects. First, it unifies scheduling with a fault-tolerance strategy. Previous work on distributable thread scheduling [13, 14] has addressed only the scheduling problem, allocating fault tolerance to functionally separate *thread integrity protocols* [13–15]. While this provides admirable separation of concerns, the scheduling and integrity operations become tightly intertwined in distributed systems where failures are prevalent. The unified approach manifested in DUA-CLA enables strong timeliness assurances in the presence of failures.

The second unique aspect of DUA-CLA stems from its *collaborative* approach to the distributed scheduling problem, rather than requiring each node independently to schedule tasks without knowledge of other nodes’ states. Global scheduling approaches wherein a single, centralized scheduler makes all scheduling decisions have been proposed and implemented. DUA-CLA, however, takes a *via media*, improving independent node scheduler’s decision-making with partial knowledge of the global system state.

Finally, we present the first experimental results for such a consensus-driven, collaborative scheduling algorithm. A fast-failure detector, consensus mechanisms, and the DUA-CLA algorithm were implemented atop DRTSJ and exercised in a network testbed to obtain real-world performance results. These performance results serve as an engineering guide to evaluate the applicability of this approach in actual systems.

Little work has been done on collaborative distributed scheduling for real-time systems. The RT-CORBA 2.0 specification [5] envisions such an approach, enumerating it as the third of its four “cases” for distributed scheduling. Poledna, et. al. consider a consensus-based sequencing approach for operations on replicas to ensure consistency [16]. In a similar vein, Gammar and Kammoun present a consensus algorithm for ensuring database properties such as serializability and consistency in real-time transactional systems [17]. None of these directly address the question of end-to-end causal activity scheduling.

The remainder of this paper presents the DUA-CLA algorithm including the design rationale and objectives. We then establish the theoretical properties, providing proofs where appropriate. Finally, we present our implementation and experimental environment. We conclude with thoughts on the future direction of this work.

2 The DUA-CLA Algorithm

2.1 Models

Distributable Thread Abstraction. Threads execute in local and remote objects by location-independent invocations and returns. A thread begins its execution by invoking an object operation. The object and the operation are specified when the thread is created. The portion of a thread executing an object operation is called a *thread segment*. Thus, a thread can be viewed as being composed of a concatenation of thread segments.

A thread’s initial segment is called its *root* and its most recent segment is called its *head*. The head of a thread is the only segment that is active. A thread can also be viewed as being composed of a sequence of *sections*, where a section is a maximal length sequence of contiguous thread segments on a node. A section’s first segment results from an invocation from another node, and its last segment performs a remote invocation. Further details of the thread model can be found in [5, 6, 14].

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., [18]). 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 $\mathbf{T} = \{T_1, T_2, T_3, \dots\}$. The set of sections of a thread T_i is denoted as $[S_1^i, S_2^i, \dots, S_k^i]$.

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 in this paper 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 and Failure Models Our system and failure models follow that of [19]. We consider a system model where a set of processing components, generically referred to as *nodes*, denoted by the totally-ordered set $\Pi = \{1, 2, \dots, n\}$, 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 DUA-CLA 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 [20] 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 [21] $d \leq D$ is assumed to be bounded. Similar to [19], 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 $t + d$.

2.2 Scheduling Objectives

Our primary objective is to design a thread scheduling algorithm that will maximize the total utility accrued by all the threads as much as possible. Further, the algorithm must provide assurances on the satisfaction of thread termination times in the presence of (up to f_{max}) crash failures. Moreover, the algorithm must exhibit the best-effort property of UA algorithms (described in Section 1).

2.3 Rationale and Design

In the absence of crash failures, there is no compelling motivation for nodes to collaborate for constructing a system-wide schedule.¹ 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.

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

¹One 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.

while that decision is being made. We formulate this problem as a *consensus* problem [22] 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.² 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 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.

2.4 Algorithm Description

The DUA-CLA algorithm that we present is derived from Aguilera *et. al*’s time-optimal, early-deciding, uniform consensus algorithm in [19]. A pseudo-code description of DUA-CLA on each node i is shown in Algorithm 1.

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.

²A “scheduling event” is an event that invokes the scheduling algorithm.

Algorithm 1: DUA-CLA: Code for each node i

```
2 input:  $\sigma_r^i$ ; output:  $\sigma_i$ ; //  $\sigma_r^i$ : unordered ready queue of node  $i$ 's sections;  $\sigma_i$ : schedule
4 Initialization:  $\Sigma_i = \emptyset$ ;  $\omega_i = \emptyset$ ;  $max_i = 0$ ;
6  $\sigma_i = \text{ConstructLocalSchedule}(\sigma_r^i)$ ;
8 send( $\sigma_i, i$ ) to all;
10 upon receive ( $\sigma_j, j$ ) until  $2D$  do // After time  $2D$ , consensus begins
12    $\Sigma_i = \Sigma_i \cup \sigma_j$ ;
14  $\omega_i = \text{DetermineSystemWideFeasibleThreadSet}(\Sigma_i)$ ;
16 upon receive ( $\omega_j, j$ ) do
18   if  $j > max_i$  then  $max_i = j$ ;  $\omega_i = \omega_j$ ;
20 at time  $(i - 1)d$  do
22   if suspect  $j$  for any  $j : 1 \leq j \leq i - 1$  then
24      $\omega_i = \text{UpdateFeasibleThreadSet}(\Sigma_i)$ ;
26     send( $\omega_i, i$ ) to all;
28 at time  $(j - 1)d + D$  for every  $j : 1 \leq j \leq n$  do
30   if trust  $j$  then decide  $\omega_i$ ;
32  $\text{UpdateSectionSet}(\omega_i, \sigma_r^i)$ ;
34  $\sigma_i = \text{ConstructLocalSchedule}(\sigma_r^i)$ ;
36 return  $\sigma_i$ ;
```

When invoked, a node i first constructs the local section schedule by invoking `ConstructLocalSchedule()`. 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. 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. 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 . 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 first computes a new ω_i using `UpdateFeasibleThreadSet()` (see Algorithm 2), then sends (ω_i, i) to all nodes. 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 $D > d$.

Algorithm 2: UpdateFeasibleThreadSet

```
2: input:  $\omega_i$ ; output:  $\omega'_i$ ; //  $\omega'_i$ : feasible section set with sections on failed nodes removed
4: initialize:  $\omega'_i = \omega_i$ 
6: for each section  $S_j^t \in \omega_i$  do
8:   if suspect  $j$  then  $\omega'_i = \omega'_i \setminus S_j^t$ ;
10: return  $\omega'_i$ ;
```

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 .

At times $(j - 1)d + D$ for $j = 1, \dots, n$, 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 .³

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.

³If 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 begins consensus before it executes the following asynchronous **at time** calls (similar to [19]).

2.5 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 $i = 1, 2, \dots$, 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 exist 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 3: `ConstructLocalSchedule()`

```

2: input:  $\sigma_r$ ; output:  $\sigma$ ;
4: Initialization:  $t := t_{cur}$ ,  $\sigma := \emptyset$ ;
6: for each section  $S_i \in \sigma_r$  do
8:   if  $current\_time + S_i.ex > S_i.tt$  then
9:      $\perp$  abort( $S_i$ )
11:  if  $S_{j-1}^i.tt + D + S_j^i.ex > S_j^i.tt$  then
13:     $\sigma_r = \sigma_r \setminus S_i$ ;
    // remove apparently infeasible section from consideration
14:  else
16:     $\perp$   $S_i.PUD = U_i(t + S_i.ex) / S_i.ex$ ;
18:  $\sigma_{tmp} := \text{sortByPUD}(\sigma_r)$ ;
20: for each section  $S_i \in \sigma_{tmp}$  from head to tail do
22:   if  $S_i.PUD > 0$  then
24:     Insert( $S_i, \sigma, S_i.tt$ );
26:     if Feasible( $\sigma$ )  $\neq$  false then
28:        $\perp$  Remove( $S_i, \sigma, S_i.tt$ );
30:   else break;
32: return  $\sigma$ ;

```

Algorithm 3 describes the local section scheduling algorithm. When invoked at time t_{cur} , the algorithm first checks the feasibility of the sections. First, if the earliest conceivable execution (the current time) of segment will still miss the termination time, the algorithm aborts the section. If the earliest predicted completion time of a section is later than its termination time, it is removed from consideration. This removal is temporary, and the segment will be reconsidered in subsequent scheduling events so that it may appear in future schedules due to failure or early completion of other threads. Otherwise, the algorithm calculates the section's PUD.

The sections are then sorted by their PUDs. In each step of the *for*-loop, 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. If σ becomes infeasible, S_i is removed. After examining all sections, the ordered list σ is returned. The function `Feasible()` determines the feasibility of a schedule. A schedule σ is feasible if the predicted completion time of each section $S_i \in \sigma$ does not exceed S_i 's termination time. For explicit pseudo-code for a linear-time implementation, see Algorithm 3 in [11].

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

3 Algorithm Properties

We first describe DUA-CLA’s timeliness property under crash-free runs:

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

Proof: From the discussion in Section 2.5, if a node is underloaded, Algorithm 3 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: DUA-CLA 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 [19].

Theorem 3: DUA-CLA’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 1-1 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 1.

Lines 1–1 has an actual time cost of $D + fd$ and will involve $(f + 1)n$ messages [19]. Line 1, 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: DUA-CLA is a time-optimal, early-deciding algorithm that achieves consensus on the system-wide execution-eligible thread set.

Proof: From Theorem 3, DUA-CLA decides in time proportional to f . From Theorem 2, the algorithm achieves consensus on the system-wide execution-eligible thread set. From [19], 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 DUA-CLA’s timeliness property in the presence of failures.

Theorem 5: If $n - f$ nodes (i.e., correct nodes) are underloaded, then DUA-CLA 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 DUA-CLA 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 DUA-CLA and other distributable thread scheduling UA algorithms including DASA [9], 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 DUA-CLA is $3D + \delta$, DASA’s and LBESA’s is δ , and that of AUA is $+\infty$.

Proof: DUA-CLA 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 DUA-CLA, since DASA’s and LBESA’s asymptotic computational costs are the same as that of DUA-CLA (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 model (e.g., [23]) and will reject T_i in favor of previously admitted threads, yielding a worst-case NBI of $+\infty$.

In order to further characterize the algorithm’s best-effort behavior in the presence of failures, we introduce definitions for *Lazy-Abort* behavior and *Schedule Safety*:

Definition 4: A collaborative distributable thread scheduling algorithm is said to *Lazy-Abort* if it delays abortion of a segment until it would be infeasible if it were the only thread in the system.

Theorem 7: DUA-CLA demonstrates *Lazy-Abort* behavior. Sections are only aborted in `ConstructLocalSchedule()` (Algorithm 3), and then only when the segment would exceed its deadline if it were executed immediately. If this is the case, the *Lazy-Abort* condition is met. Consequently, transient perceived overloads which resolve through node failures or pessimistic execution time evaluations do not cause overly-aggressive abortion of future threads.

Definition 5: A consensus-based distributable thread scheduling algorithm is said to exhibit *Schedule Safety* if it never allows the presence of a remote segment S_f in the global feasible set to render infeasible a local segment S_l if the node hosting S_f is known to have failed during consensus.

Theorem 8: DUA-CLA demonstrates *schedule-safety* in the presence of failures during the distributed scheduling event. The algorithm evaluates feasibility of local segments on node i based on the section set updated in the call to `UpdateSectionSet()` in Algorithm 1. If any nodes j with $1 < j < i$ is suspected by i , then i removes all segments on j from ω_i . (Furthermore, at time $D + fd$, all nodes will receive this reduced proposed section set.) Therefore, no locally feasible thread segments will be rendered infeasible by erroneous inclusion of segments from j .

4 Implementation Experience

A major objective of the work presented in this paper was to bridge the gap between purely theoretical consideration of algorithm properties and the practicalities of implementation. In particular, we constructed experiments to uncover the time complexity constants implicit in Theorem 3. Therefore, we compare single-node to distributed task sets in the presence of underloads as well as overloads, in failure-free as well as the $f = f_{max}$ case. Furthermore, we explore algorithm overhead by comparing well-known single-node scheduling disciplines to a degenerate case of our collaborative approach.

4.1 Implementation Environment

We implemented DUA-CLA in the DRTSJ reference implementation (DRTSJ-RI), consisting of Apogee’s Aphelion-based DRTSJ-compliant Java Virtual Machine (JVM). The DRTSJ-RI was installed on a collection of Pentium-class machines running the Ubuntu Edgy Linux distribution. Each node was supplied with a custom Linux kernel (version 2.6.17) compiled with real-time extensions.

Nodes were interconnected via 10Mbps Ethernet through a Debian Linux based dynamic switch. The dynamic switch was configured with the `netem` network emulation module [24] in order to selectively introduce controlled communication delay. Further, the Linux advanced routing and traffic control capability was employed to apportion traffic into three priority bands: Failure detector traffic, experimental control traffic, and normal communication traffic. These bands were configured to simulate communication delays consonant with the system model described in Section 2.1, resulting in particular in the relationship $D \gg d$ between common communication latency and failure detection latency.

A timer-based fast failure detector was implemented using the pure Real-Time Specification for Java API [25]. With this detector in place, the DUA-CLA algorithm was implemented on top of the DRTSJ Metascheduler. The Metascheduler is a pluggable scheduling framework developed by our team, enabling user-space thread scheduling implementations in DRTSJ [6], and previously in C on QNX [26].

One of the more challenging aspects of implementing algorithms such as DUA-CLA is the presence of explicit, asynchronous control statements such as “at time t , do...”. The `AsynchronousEvent` abstraction from RTSJ, properly employed, allowed this logic to be correctly and efficiently designed.

4.2 Experimental Setup

Our experiments took place in the testbed described above. One DRTSJ-compliant JVM was instantiated on each node. Node clocks were synchronized using Network Time Protocol (NTP) [27]. We configured the dynamic switch to insert communication delays in application communication traffic following a normal distribution. All application communication was via unreliable datagram (UDP), with reliable message delivery assured at the application layer. Additionally, we measured message loss for FD and application traffic across a range of network load conditions to ensure that reliability assumptions were not violated.

4.3 Fast Failure Detection

We implemented a naive heartbeat failure detector and tuned it for the network testbed. In our implementation, the heartbeat broadcast rate used was 1 millisecond, with an evaluation period of 3 milliseconds. The FD component is written as a very small body of code in pure RTSJ, and executed with the highest execution eligibility on the node. Furthermore, RTSJ `ScopedMemory` areas were employed to avoid any garbage collection latency.

Figure 4.3 presents our measurements of failure detector latency d and application message delay D . At no time was a failure detection latency greater than 2.98 milliseconds measured, and therefore we use $d = 2.98\text{ms}$ for the following DUA-CLA experiments. Similarly, we measured a worst-case message delay $D \approx 69.87\text{ms}$. These latencies are shown to be stable across a range of CPU utilization. We therefore claim that we have implemented a perfect, fast failure detector (given the narrow constraints of these experiments).

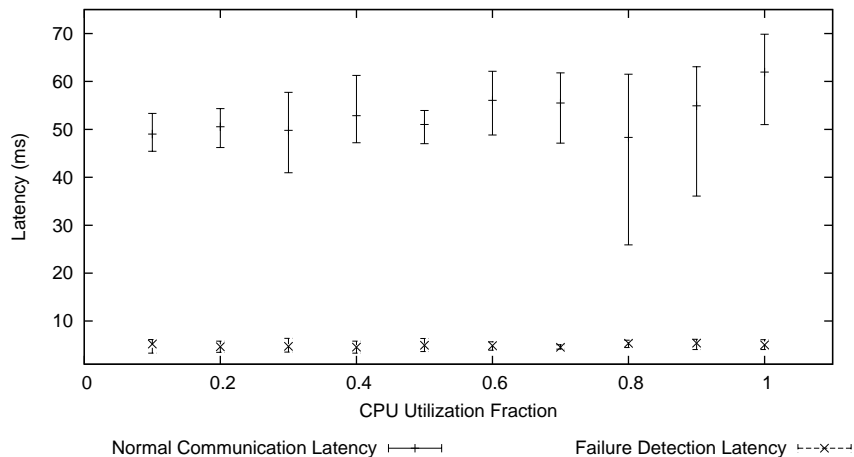


Figure 1: Failure Detection and Communication Latency vs. System Load

4.4 Scheduler Performance

Local Scheduler Performance. In order to establish a baseline for assessing the performance of our scheduler implementation, we conducted a series of experiments comparing DUA-CLA to a variety of other scheduling algorithms. In these experiments, each submitted thread consisted of a single segment to be executed on the local node. Since no remote segments appeared in the ready set, no remote scheduling event was triggered and DUA-CLA is functionally equivalent to Clarke’s DASA algorithm.

Figure 2 illustrates the deadline satisfaction performance of several UA and non-UA scheduling policies. We use the metric Deadline Satisfaction Ratio (DSR), the ratio of jobs which satisfy their deadline to the total number of jobs released. In the case of this non-distributed experiment, a job is exactly equivalent to a thread segment. For these plots, a collection of 5 periodic threads were created with relatively-prime periods and random phase offsets. The mean execution time for each job release was then varied to produce a processor demands ranging from 0 to 200%.

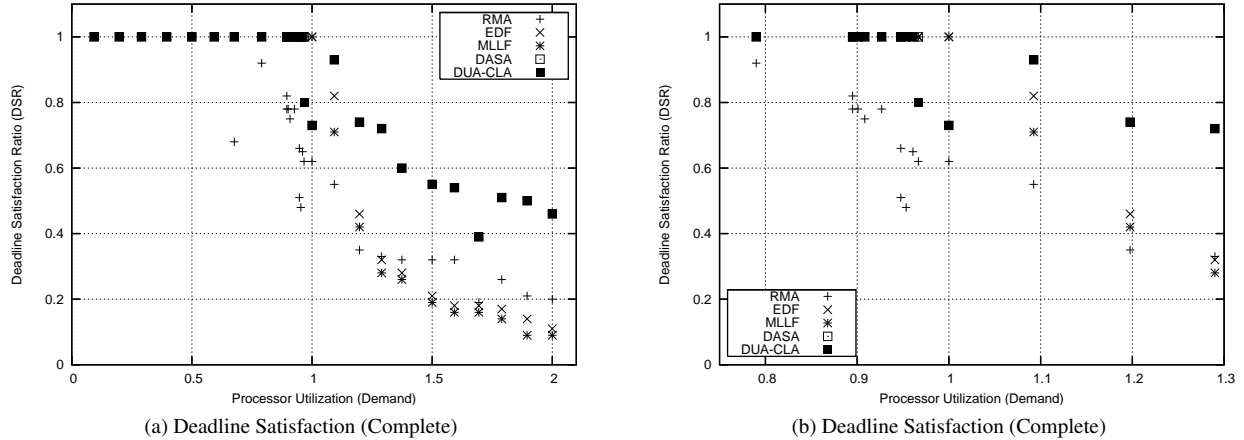


Figure 2: Local Scheduler Deadline Satisfaction

The schedulers presented here include Rate Monotonic Analysis (RMA), Earliest Deadline First, Modified (Quantum-Free) Least-Laxity First (MLLF), Dependent Activity Scheduling Algorithm (DASA), and DUA-CLA. Of these, only DASA and DUA-CLA are utility accrual algorithms. Each of these algorithms was implemented in the DRTSJ Metascheduler, and each was run with an identical task set for each utilization. The mean task execution times for these experiments ranged from tens of milliseconds (at low utilization) to hundreds of milliseconds (at high utilization).

Figure 2b provides a detailed look at the “deadline-miss load” region from Figure 2a, the processor utilization range at which the scheduling policies begin missing activity deadlines. Theoretically, each of the schedulers shown (with the exception of RMA) should obtain 100% DSR up to 100% load. However, due to the overhead of the Metascheduler, Java virtual machine, operating system, and the execution time of the scheduling policy itself, activities miss deadlines at lower CPU utilizations. Understanding the size of this overhead as we consider more complex scheduling policies is critical to engineering systems which appropriately trade off scheduling “intelligence” against the additional overhead incurred by more complex policies.

The reader should note that, for the local-only scheduling case, DUA-CLA is almost indistinguishable from DASA. This is because these two algorithms use identical policies for sequencing local activities and assessing their feasibility. The slight overhead penalty incurred in DUA-CLA to determine that no consensus decision is required is less than 1% across the range of task profiles we considered.

Figure 3a captures the performance of these schedulers when measured against the UA metric Accrued Utility Ratio (AUR). AUR is the ratio of the accrued utility (the sum of the U_i for all completed jobs) to the utility available (sum of U_i for all released jobs.) Since we have chosen unit-downward step TUFs for these experiments, the AUR and DSR are similar metrics, with AUR appearing as a weighted form of the DSR, with weights U_i . The reader will note that, while Figure 2 indicates that the non-UA policies like RMA sometimes outperform DASA and DUA-CLA in the DSR metric during overloads, Figure 3a shows us that this is because RMA is dropping the “wrong” tasks, while the UA policies shed load favoring completion of high-utility tasks. It is precisely this behavior we wish to explore in the distributed case, in particular understanding the additional overhead incurred.

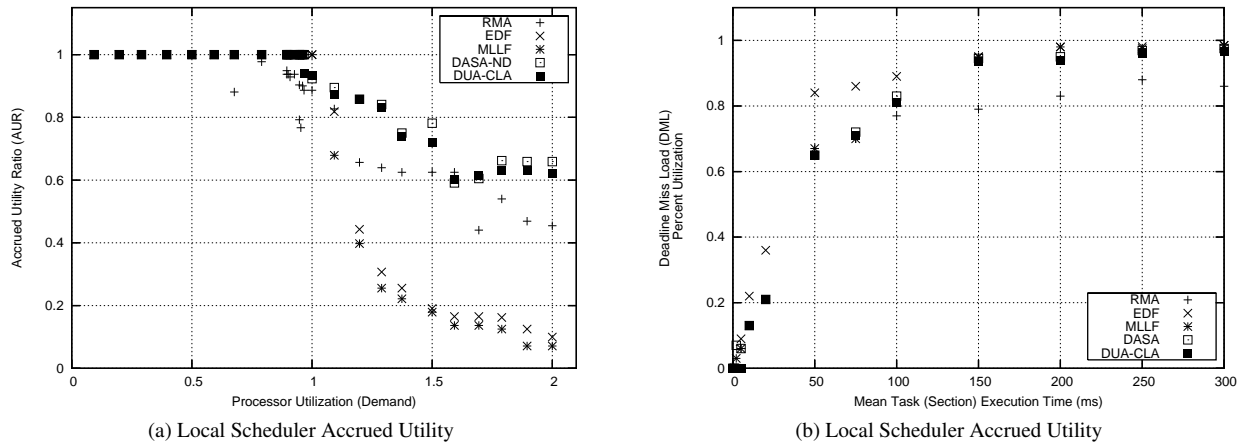


Figure 3: Local Scheduler Performance

The final local experiment conducted is shown in Figure 3b, which more clearly characterizes the scheduler overhead for each of the policies by measuring their Deadline Miss Load (DML) for a range of task execution times. For each data point illustrated, we fixed an execution time (shown on the plot’s x -axis) for every job during a single experimental scheduler run. We then varied the periods of a collection of periodic threads, measuring the resulting utilization and deadline satisfaction performance. The DML (shown on the y -axis) is the greatest utilization for which the indicated scheduling policy was able to meet all deadlines. Again, note that the theoretical behavior during underload for each policy shown is for a DML of 1.0 — in other words, these policies should never miss a deadline until the CPU is saturated.

Practically, however, the DML varies with (mean) job execution time. At a fixed utilization, as the execution time of the jobs decreases towards the same order of magnitude as the scheduler execution time plus the platform overhead, the policy is simply unable to make a scheduling decision fast enough to schedule a new job.

Distributed Scheduler Performance. Our final set of experiments sought to establish the concrete behavior of the DUA-CLA algorithm for qualitative comparison to local scheduling approaches, for validation of the theoretical results above, and to investigate execution timescales for which consensus thread scheduling is appropriate.

These experiments included scheduler runs with three segment periodic threads. In this formulation, every thread originates on a common origin node with a short segment (1). Each makes an invocation onto one of many server nodes to execute segment (2), then returns to the origin node to complete in segment (3). The periods of these threads are fixed, and the execution times are varied to produce the range of utilizations in Figure 4. In these distributed experiments, the utilization axis represents the *maximum* utilization experienced by any node at any time.

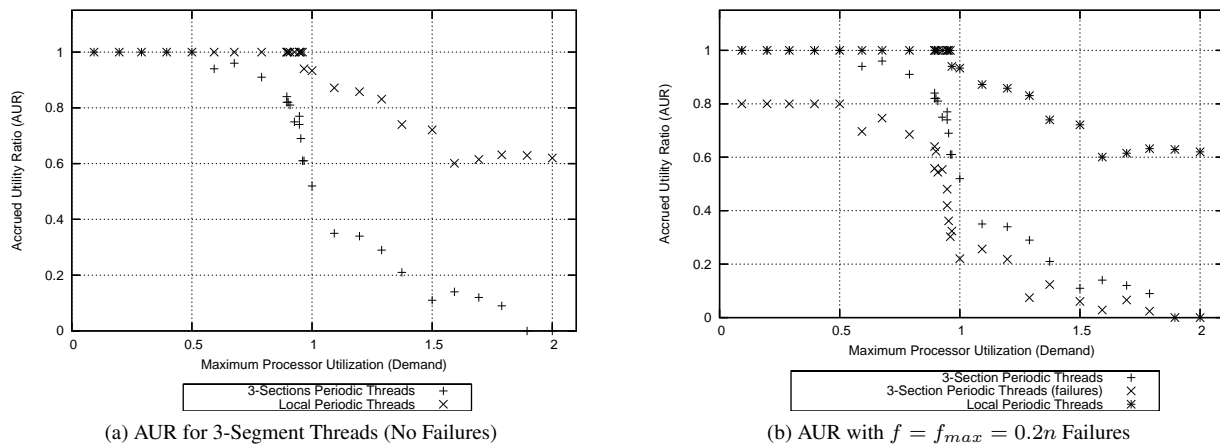


Figure 4: Distributed Scheduler Performance

In Figure 4a, we compare the AUR of a collection of one-segment (local) threads to a collection of three-segment threads. As can be seen from the plot, the penalty incurred by collaboration is significant, but the scheduling policy continues to accrue utility through 1.8 fractional utilization. Furthermore, Theorem 1 is borne out by the underloaded portion of Figure 4a, modulo scheduling overhead. This overhead is explored in more detail in the discussion of Figure 4.4.

The behavior of DUA-CLA in the presence of failures is shown in Figure 4b, wherein we programmatically fail $f_{m,ax}$ nodes, amounting in this experiment to 20% of the total nodes. Again, the performance of the scheduler suffers, but as shown in Theorem 5, our implementation meets the termination times for all threads remaining on correct nodes.

Finally, we investigate the overhead incurred by DUA-CLA across a selection of mean task execution times. Figure 4.4 demonstrates the expected penalty paid in terms of DML for conducting collaborative scheduling. Annotations show D , $2D$, and $3D$ communication step boundaries, and it is clear that the DML for tasks with execution times less than $3D$ suffers significantly because this is the minimal communication delay required to accept a thread’s segments for execution.

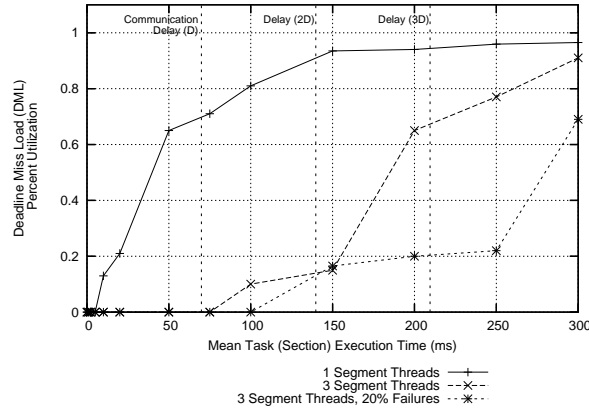


Figure 5: Deadline Miss Load for DUA-CLA

5 Conclusions and Future Work

The preliminary investigation of consensus-driven collaborative scheduling approach described in this work may be extended in a variety of ways. In particular, algorithmic support for shared resources, deadlock detection and resolution, and quantitative assurances during overload represent worthwhile theoretical questions. Furthermore, improved implementations investigating real-world behavior under failures and with non-trivial abort handling are suggested by the results presented here. An exhaustive look at the practical message complexity would enable broad-based analysis of algorithm design and implementation trade-offs between time complexity and overload schedule quality.

References

- [1] CCRP, “Network centric warfare.” <http://www.dodccrp.org/ncwPages/ncwPage.html>.
- [2] J. D. Northcutt and R. K. Clark, “The Alpha operating system: Programming model,” Archons Project Technical Report 88021, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1988.
- [3] B. Ford and J. Lepreau, “Evolving Mach 3.0 to a migrating thread model,” in *USENIX Technical Conference*, pp. 97–114, 1994.
- [4] The Open Group, *MK7.3a Release Notes*. Cambridge, Massachusetts: The Open Group Research Institute, October 1998.
- [5] OMG, “Real-time CORBA 2.0: Dynamic scheduling specification,” tech. rep., Object Management Group, September 2001.
- [6] J. Anderson and E. D. Jensen, “The distributed real-time specification for Java: Status report,” in *JTRES*, 2006.
- [7] H. Chetto and M. Chetto, “Some results of the earliest deadline scheduling algorithm,” *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 466–473, 1989.
- [8] E. D. Jensen *et al.*, “A time-driven scheduling model for real-time systems,” in *RTSS*, pp. 112–122, Dec. 1985.
- [9] R. K. Clark, *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, August 1990.
- [10] C. D. Locke, *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, CMU, 1986. CMU-CS-86-134.
- [11] 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 *Proceedings of SEUS 2007*, May 2007. (To appear.) Available at <http://www.real-time.ece.vt.edu/seus07.pdf>.
- [12] K. Han, B. Ravindran, and E. D. Jensen, “RTMG: Scheduling distributable real-time threads in large-scale, unreliable networks with low message overhead.” Also submitted for publication at RTCSA07, in parallel with this paper, April 2007.
- [13] E. Curley, J. S. Anderson, *et al.*, “Recovering from distributable thread failures with assured timeliness in real-time distributed systems,” in *SRDS*, pp. 267–276, 2006.
- [14] J. D. Northcutt, *Mechanisms for Reliable Distributed Real-Time Operating Systems — The Alpha Kernel*. Academic Press, 1987.
- [15] J. Goldberg, I. Greenberg, *et al.*, “Adaptive fault-resistant systems (chapter 5: Adaptive distributed thread integrity),” Tech. Rep. csl-95-02, SRI International, January 1995. <http://www.csl.sri.com/papers/sri-csl-95-02/>.
- [16] S. Poledna, A. Burns, A. Wellings, and P. Barrett, “Replica determinism and flexible scheduling in hard real-time dependable systems,” *IEEE Transactions on Computing*, February 2000.
- [17] S. M. Gammar and F. Kamoun, “A comparison of scheduling algorithms for real time distributed transactional systems,” in *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, 1997*, pp. 257–261, October 1997.
- [18] D. P. Maynard, S. E. Shipman, *et al.*, “An example real-time command, control, and battle management application for Alpha,” Tech. Rep. Archons Project Technical Report 88121, CMU CS Dept, December 1988.
- [19] M. K. Aguilera, G. L. Lann, and S. Toueg, “On the impact of fast failure detectors on real-time fault-tolerant systems,” in *DISC*, pp. 354–370, Springer-Verlag, 2002.
- [20] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *JACM*, vol. 43, no. 2, pp. 225–267, 1996.
- [21] W. Chen, S. Toueg, and M. K. Aguilera, “On the quality of service of failure detectors,” *IEEE Transactions on Computers*, vol. 51, pp. 561–580, May 2002.
- [22] N. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [23] A. Bestavros and S. Nagy, “Admission control and overload management for real-time databases,” in *Real-Time Database Systems: Issues and Applications*, ch. 12, Kluwer Academic Publishers, 1997.
- [24] “Netem Wiki.” <http://linux-net.osdl.org/index.php/Netem>.
- [25] “Real-time specification for Java.” <http://rtsj.org>.
- [26] P. Li, B. Ravindran, *et al.*, “A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems,” *IEEE Trans. Software Engineering*, vol. 30, pp. 613 – 629, Sept. 2004.
- [27] D. L. Mills, “Improved algorithms for synchronizing computer network clocks,” *IEEE/ACM TON*, vol. 3, pp. 245–254, June 1995.