Recovering from Distributable Thread Failures with Assured Timeliness in Real-Time Distributed Systems

Edward Curley[⋆], Jonathan Anderson[⋆], Binoy Ravindran[⋆], and E. D. Jensen[‡]

[⋆]ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{alias,andersoj,binoy}@vt.edu

[‡]The MITRE Corporation
Bedford, MA 01730, USA
jensen@mitre.org

Abstract

We consider the problem of recovering from failures of distributable threads with assured timeliness. When a node hosting a portion of a distributable thread fails, it causes *orphans* — i.e., segments of distributable threads that are disconnected from the thread's root. We consider a termination model for recovering from such failures, where the orphans must be detected and aborted, resources held by them must be released and rolled back to safe states, and exceptions must be delivered to farthest, contiguous surviving thread segment from where execution can be resumed. Since distributable threads are subject to time constraints in real-time distributed systems, such recovery must be conducted with assured timeliness. Toward this, we present 1) a real-time scheduling algorithm called AUA, and 2) a distributable thread integrity protocol called *TP-TR*. We show that *AUA* and *TP-TR* bound the orphan cleanup and recovery time (thereby bounding thread starvation durations), maximize total thread accrued timeliness utility, and satisfy thread mutual exclusion constraints. We implement *AUA* and *TP-TR* in a real-time middleware that supports distributable threads. Our experimental studies with the implementation validate the algorithm/protocol's time-bounded recovery property and confirm their effectiveness.

Index Terms

distributable thread, thread maintenance and recovery, time/utility function, utility accrual scheduling

I. Introduction

Many distributed systems are most naturally reasoned about in terms of asynchronous concurrent sequential flows of execution within and among objects. The *distributable thread* programming model supported in OMG's recent Real-Time CORBA 1.2 standard (abbreviated here as RTC2) [1] and Sun's upcoming Distributed Real-Time Specification for Java (DRTSJ) standard [2] directly provides that as a first-class abstraction. Distributable threads first appeared in the Alpha OS [3], [4] and later in Alpha's descendant, the MK7.3 OS [5].

A distributable thread is a single thread of execution with a globally unique identifier that transparently extends and retracts through local and remote objects. Thus, a distributable thread is an end-to-end control flow abstraction, with a logically distinct locus of control flow movement within/among objects and nodes. In the rest of the paper, we will refer to distributable threads as *threads* except as necessary for clarity.

A thread carries its execution context as it transits node boundaries, including its scheduling parameters (e.g., time constraints, execution time), identity, and security credentials. Hence, threads require that Real-Time CORBA's *Client Propagated* model be used, not the *Server Declared* model. The propagated thread context is used by node schedulers for resolving all node-local resource contention among threads such as that for node's physical (e.g., CPU, I/O) and logical (e.g., locks) resources, and for scheduling threads to optimize system-wide timeliness. Thus, threads constitute the abstraction for concurrency and scheduling. Figure 1 cited from [1] shows the execution of threads.

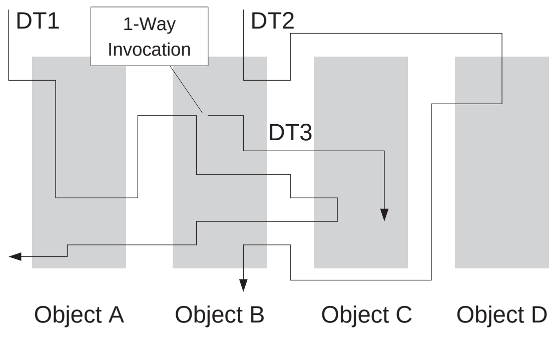


Fig. 1. Distributable Threads

The Real-Time CORBA specification envisions four distributed scheduling 'cases', summarized in Table I. This paper explicitly supports distributed scheduling schemes corresponding to Case 1 (in the case of local use of the *AUA* protocol) and Case 2 (for distributed threads). While the Real-Time CORBA specification does not address thread integrity concerns in any detail, it might be argued that the *TP-TR* protocol discussed in this paper amounts to a form of distributed resource management (specifically in the presence of partial failures) properly classified under Case 3 or 4.

Experience with OSes and middleware that directly provide the thread abstraction show that the abstraction can reduce application development and maintenance costs. This is in contrast with situations

Case 1)	Scheduling decisions take place independently on each node.
Case 2)	Scheduling decisions take place independently on each node, subject to time constraints which propagate between nodes with application activities.
Case 3)	Scheduling decisions are made by a distributed scheduling algorithm with instances on each node. Local scheduler instances collaborate to achieve or approximate global optimality.
Case 4)	Scheduling is hierarchical, with higher-level schedulers above case 1 or 2 instances which seek to improve resource allocation decisions with some global knowledge.

TABLE I

Real-Time CORBA Distributed Scheduling Cases [1, Section 3.8]

where a similar trans-node, control flow abstraction has to be emulated using lower level abstractions such as RPC's/RMI's, OS threads, locks, etc., and end-to-end properties of the flows have to be maintained such as assuring satisfaction of end-to-end time constraints, ensuring flow integrity through distributed node and link failure detection and recovery, and ensuring system safety through distributed deadlock detection and recovery.

A. Dynamic Systems and TUF/UA Scheduling

In this paper, we focus on real-time distributed systems that operate in environments with dynamically uncertain properties. These uncertainties include transient and sustained resource overloads (due to context-dependent, activity execution times), arbitrary arrival patterns for application activities, and arbitrary node/link failure occurrences. Nevertheless, such systems' desire the strongest possible assurances on activity timeliness behavior. Another important distinguishing feature of most of these systems is their relatively long activity execution time magnitudes, compared to those of conventional real-time subsystems—e.g., in the order of milliseconds to minutes. Some examples of such dynamic systems that motivate our work (from the defense domain) include phased array radars [6]), surveillance aircrafts [7]–[9]), and network-centric warfare [10], [11]).

An activity's urgency is typically orthogonal to its relative importance—-e.g., the most urgent activity can be the least important, and vice versa; the most urgent can be the most important, and vice versa. Hence when resource overloads occur, completing the most important activities irrespective of activity urgency is often desirable. Thus, a clear distinction has to be made between urgency and importance, during overloads. During under-loads, such a distinction is not necessary, because deadline-scheduling algorithms can meet all deadlines [12].

Deadlines by themselves cannot express both urgency and importance. Thus, we consider the abstraction of time/utility functions (or TUFs) [13] that express the utility of completing an application activity as a function of that activity's completion time. We specify deadline as a binary-valued, downward "step" shaped TUF; Figure 2 shows examples. Note that a TUF decouples importance and urgency—i.e., urgency is measured as a deadline on the X-axis, and importance is denoted by utility on the Y-axis.

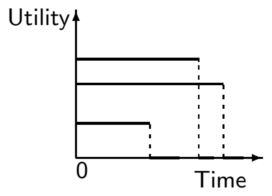


Fig. 2. Example Step TUFs

When time constraints are expressed with TUFs, the scheduling optimality criteria are typically to maximize the accrued activity utility—e.g., maximizing the sum of the activities' attained utilities. Such criteria are called Utility Accrual (UA) criteria, and sequencing (scheduling, dispatching) algorithms that optimize UA criteria are called UA sequencing algorithms (see algorithms in [14]). RTC2 has IDL interfaces for the UA scheduling discipline, besides others.

UA criteria directly facilitate adaptive behavior during resource overloads, when (optimally or sub-optimally) completing activities that are more important than those which are more urgent is often desirable. For example, UA algorithms that maximize summed utility typically meet all activity deadlines when sufficient CPU time is available for doing so [15]–[17]. Further, when overloads occur, such algorithms favor activities that are more important (from whom greater utility can be accrued) than those which are more urgent.

B. Our Contributions: Time-Bounded Thread Maintenance and Recovery

When nodes transited by distributable threads fail, this can cause threads that span the nodes to break by dividing them into several pieces. Segments of a thread that are disconnected from its node of origin (called the thread's root), are called orphans. For providing the abstraction of a continuous reliable thread, orphan segments of the thread must be detected and aborted, resources held by them must be released and rolled back to safe states, and a failure exception must be delivered to the farthest execution point of the surviving portion of the thread — i.e., the farthest contiguous thread segment from the thread's root. We focus on such a termination exception handling model as that is consistent with most concurrent programming paradigms (e.g., Ada, Java).[1]

When threads are subject to time constraints, orphan cleanup and removal must be done in a timely manner. For example, cleanup and removal of orphans of a failed low-urgent/important thread must

[1]Under a continuation model, the orphan segments are simply allowed to continue execution. A discussion of termination versus continuation models is beyond the scope of this paper.

cause as minimum interference to high-urgent/important threads as possible. On the other hand, if orphans of a failed low-urgent/important thread hold resources which are blocking high-urgent/important threads, then the cleanup activity must have execution eligibility that reflect the urgency/importance of the blocked threads. Furthermore, once a failure occurs, the time interval between detection of the thread failure and notification of the failure exception to the farthest, contiguous surviving thread segment must be bounded. If this time interval is unbounded, it can potentially cause starvation—e.g., threads blocked on resources held by orphans can never be unblocked since those orphans are never cleaned-up. Thread breaks are countered and thread integrity is maintained through thread integrity protocols.

In this paper, we present a UA scheduling algorithm *Abort-assured Utility Accrual scheduling algorithm* (or AUA), and a thread integrity protocol called *Thread Polling with Time-bounded Recovery* (or *TP-TR*) that achieve these objectives. The algorithm and the protocol consider a programming model, where real-time application activities are programmed using distributable threads that are subject to (end-to-end) time constraints specified using TUFs. Activities may arrive at arbitrary times, thus arbitrarily spawning threads. Threads may span nodes that are subject to arbitrary crash failures. Upon arrival at a node, threads are assumed to present execution time estimates of normal and abort code (or exception handler code) segments of the thread at that node to the node scheduler.

While execution time estimates of normal code can be violated at run-time (e.g., due to context dependence), causing overloads, that of abort code cannot be, as they are assumed to be non-context-dependent and relatively short (compared with normal code). Threads may share non-CPU resources that are subject to mutual exclusion constraints.

For such an application and system model, the algorithm/protocol objective is to maximize the total thread accrued utility and bound orphan cleanup and recovery time, while satisfying mutual exclusion constraints. We show that *AUA* achieves optimal total accrued utility during (the special case of) underloads and no failures, and maximizes total utility during overloads and failures. We establish that the algorithm in conjunction with *TP-TR*, always bound cleanup and recovery times (bounding thread starvation), and respect mutual exclusion. Further, we implement *AUA* and *TP-TR* in an RTC2-like real-time middleware and conduct experimental studies. Our initial measurements from the implementation validate *AUA*/*TP-TR*'s properties and confirm their effectiveness.

Like UA algorithms, thread integrity protocols have been developed in the past—e.g., Alpha's Thread Polling protocol [18], [19], the Node Alive protocol [20], and their adaptive versions [20]. However, to the best of our knowledge, no combination of UA algorithm and thread integrity protocol exist that provide end-to-end time-bounded cleanup and recovery, which is precisely what our work does. Thus, the paper's central contribution is the AUA algorithm and the *TP-TR* protocol that provide time-bounded cleanup and recovery.

The rest of the paper is organized as follows: In Section II, we discuss the models of our work and state the algorithm/protocol objectives. Section III presents the *AUA* algorithm, and section IV presents the *TP-TR* protocol. In Section V, we discuss our implementation experience. We conclude the paper and identify future work in Section VI.

II. Models and Algorithm/Protocol Objectives

Hybrid approach has been explored in other contexts, for example by Nagy and Bestavros in the context of soft-deadline transactions. [21]

A. *Distributable Thread Abstraction*

We consider RTC2/DRTSJ's distributable threads as our programming and scheduling abstraction. Details of this model can be found in [1]; here we summarize and provide an overview.

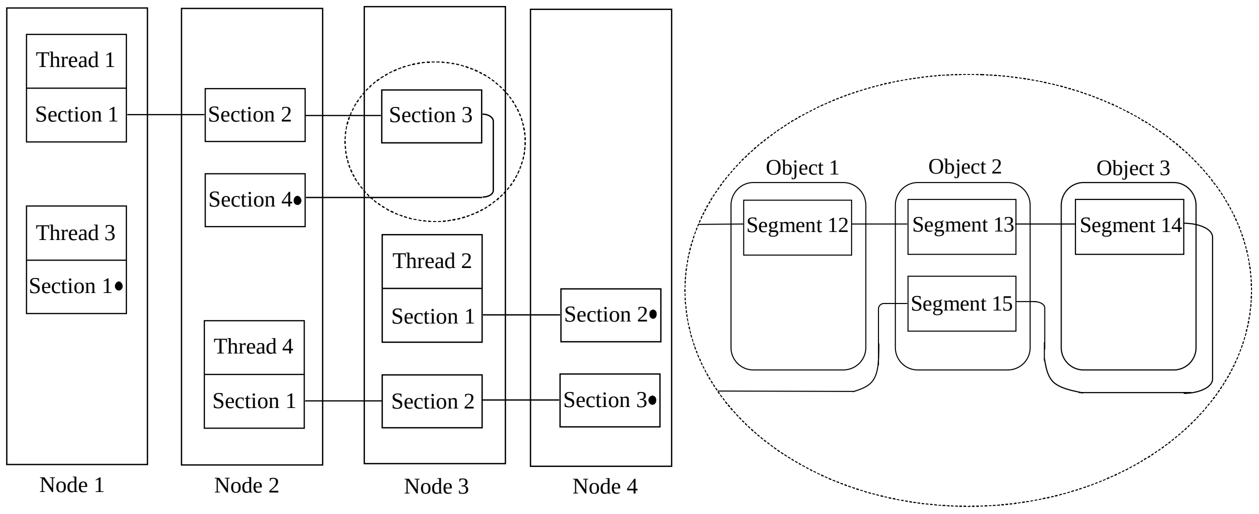
Distributable threads execute in local and remote objects by location-independent invocations and returns. A distributable thread begins its execution by invoking an object operation. The object and the operation are specified when the thread is created. The portion of a thread executing an object operation 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. The first segment in the section results from an invocation from another node and the last segment in the section performs a remote invocation. Figure 3 cited from [20] illustrates threads, sections, and segments.

The application is thus comprised of a set of threads, denoted $\mathbf{T} = \{T_k : 1 \leq k \leq n\}$.

We specify the time constraint of each thread using a TUF. The TUF of a thread T_i is denoted as $U_i\left(\right)$. Thus, thread T_i's completion at a time t will yield an utility $U_i\left(t\right)$. Though TUFs can take arbitrary shapes, here we focus on step-shaped functions as shown in Figure 2.

Each TUF U_i has an initial time I_i and a termination time X_i. Initial time is the earliest time for which the function is defined, while termination time denotes the last point that a TUF crosses the *x-axis*. For a step downward TUF, the critical time is its discontinuity point. We assume that $U_i\left(t\right) > 0, \forall t \in \left[I_i, X_i\right]$ and $U_i\left(t\right) = 0, \forall t \notin \left[I_i, X_i\right], i \in \left[1, n\right]$.



(a) Four Threads (names mark roots and dots mark heads) (b) Thread Segments in (encircled) Thread Section

Fig. 3. Distributable Threads, Segments, and Sections

B. System and Failure Models

We consider a system model, where a set of processing components, generically referred to as *nodes*, are interconnected via a network. Each node executes thread segments. The order of executing segments on a node is determined by the scheduler residing at the node. We consider RTC2's *Case 2* approach [1] for thread scheduling. According to this approach, node schedulers use the propagated thread scheduling parameters and independently schedule thread segments on respective nodes to optimize the system-wide timeliness optimality criterion. Thus, scheduling decisions made by a node scheduler are independent of other node schedulers. Though this results in approximate, global, system-wide timeliness, RTC2 supports the approach, due to its simplicity and capability for coherent end-to-end scheduling. The approach's effectiveness is illustrated in Alpha OS [4] and Tempus middleware [22].[2]

We consider a Local Area Network model (e.g., a single broadcast domain), where nodes are interconnected through a switch. We presume the existence of a reliable message transport with worst case message delivery latency D. Not all aspects of the protocol require reliable messaging. Message packets that are generated when threads invoke remote object operations, will contend for the network links. Such contentions must be resolved and packets must be scheduled on network links using a packet scheduling algorithm. We do not consider any particular algorithm for scheduling packets; *AUA* and *TP-TR* are independent of any such algorithm.

We denote the set of nodes as $P_i \in P, i \in [1, m]$. We assume that all node clocks are synchronized

[2]RTC2 also describes Cases 1, 3, and 4, which describe non real-time, global and multilevel distributed scheduling, respectively [1]. However, RTC2 does not support Cases 3 and 4.

using a protocol such as [23]. We consider an arbitrary, crash failure model for the nodes—i.e., any node can fail at any time and when it does so, it simply halts.

III. The *AUA* Algorithm

A. Rationale

In order to attain bounded recovery time for distributed threads, it is necessary to have a scheduling algorithm which guarantees a bound on the time required by each orphaned distributed thread section to detect and conduct cleanup operations. Without this guarantee, it would be possible for a broken thread to leave the system in an unsafe state. In particular, it would be possible for a single distributed thread to have multiple, uncoordinated points of execution for an unbounded amount of time. In order to facilitate this guarantee, we have developed the *AUA* scheduling algorithm, described below.

The *AUA* scheduling algorithm is a hybrid approach, providing traditional hard real-time guarantees for the execution of those blocks of code designated as cleanup handlers. As such, traditional hard real-time analysis techniques may be applied to this (typically small) subset of the application code. In particular, these guarantees are exploited by the *TP-TR* thread integrity protocol presented in Section IV.

For the remainder of the application code, the *AUA* scheduling algorithm provides best-effort utility accrual scheduling similar to that presented extensively in the literature. *AUA* traces its lineage to the DASA scheduling algorithm introduced by Clark [24], and is equivalent to DASA if no abort handlers are introduced.

B. Algorithm Overview

When a thread segment/handler pair is introduced to the system, the scheduler first checks to see if the handler's execution can be guaranteed. If not, the thread segment/handler pair is rejected and no new schedule is created. If a new handler is accepted, its last-chance time (LCT) to commence execution is calculated by subtracting its WCET from its deadline, allowing the scheduler to plan for the last moment at which the abort handler can be guaranteed to execute to completion.

At a scheduling event, all handlers are first inserted into an EDF-ordered schedule. Only after it has been verified that each thread segment's abort handler is feasible does *AUA* proceed to the best-effort optimization step. Thread segments are inserted into this schedule using the *DASA* algorithm, considering thread segments in decreasing order of their value density, and inserting them into a deadline-ordered schedule if they are feasible. Once the schedule is created, the first entity (either task or handler) in the schedule is chosen for execution.

Observation AUA-1: *If no abort handlers are submitted to the schedule, AUA is identical to DASA. Consequently, the properties guaranteed by DASA in underload hold. In particular, AUA is equivalent to the known-optimal Earliest Deadline First (EDF) scheduling discipline if there are no abort handlers and the system is in underload. This useful property is borne out in our experiments below.*

When a handler's LCT arrives without the handler's task having completed, the scheduler automatically wakes up and schedules the handler, thus guaranteeing that the handler is completed by its deadline.

Scheduling events in *AUA* include the arrival of a thread/handler pair, the completion of a thread, the completion of a handler, a resource request, a resource release, and the arrival of a handler's last-chance time (LCT). For clarity, we present only those events which directly impinge on AUA's performance. See [22] for a thorough description of resource request/grant event processing in the *Metascheduler*.

In order to describe *AUA*, we introduce the notation given in Table II. A high-level description of the *AUA* scheduling algorithm is presented in Algorithm 1, which we discuss below.

When the algorithm is invoked at time t_{cur}, it first computes a deadline ordered schedule of all the abortion handlers accepted into the system. Then, depending on the scheduling event, it will do one of three things: attempt to add a handler to the system, remove a handler from the system, or schedule the handler with the earliest deadline for execution. The scheduling events associated with each action are stated in Algorithm 1.

The setLCT() function sets a timer that will wake the scheduler when the next LCT arrives. The dependencies and value density (VD) of each thread are then calculated using getDep() and computeVD(), respectively. The function sortByVD() then uses the value density to create a list of all the threads in the system sorted in non-increasing order by VD. *AUA* goes through this list in order and attempts to insert each thread into a feasible, deadline ordered list using procedure insertByEDF(). Finally, *AUA* returns the thread T_{exe}, which is the task within the feasible schedule with the earliest deadline.

T_r	current set of unscheduled threads accepted into the system.
T_c	current set of handlers accepted into the system.
T_{rnew}	new thread arriving in the scheduling event
T_{cnew}	new handler arriving in the scheduling event
T_i.DL	the thread's deadline for $T_i \in T_r$
T_i.ExecTime	the thread's remaining execution time
T_i.Value	potential utility gained the if the thread completes before its deadline
T_i.Dep	The task that T_i is dependent on
σ	the current ordered schedule
$\sigma(i)$	denotes the thread occupying the ith position in the schedule σ
reqResource(T_i)	returns the resource requested by T_i
owner(R)	returns the thread that is currently holding resource R
headOf(σ)	returns the first thread in schedule σ
sortByVD(σ)	returns a new schedule sorted by non-increasing value density (VD)
insert(T, σ, i)	inserts thread T in the ordered list σ at the position indicated by index i; if there are already entries with the index i, T is inserted before them. After insertion, the index of T in σ is i
remove(T, σ)	removes T from the ordered list σ. If T is not in σ no action is taken.
feasible(σ)	returns a boolean value indicating schedule σ's feasibility. For σ to be feasible, the predicted completion time of each thread in σ must never exceed its deadline.
setLCT(t)	sets a timer to wake up the scheduler at time t
copySchedule(σ)	makes a copy of schedule σ

TABLE II

Variables and Operations used in AUA

Algorithm 1: AUA Algorithm

```
Data: $T_r$, $T_c$, $event$
Result: selected thread to execute, $T_{exe}$
1  t ← $T_{cur}$ ;
2  $\sigma \leftarrow \emptyset$ ;
3  for $T_i \in T_c$ do  $\sigma \leftarrow$ insertByEDF($T_i$, $\sigma$) ;
4  switch $event$ do
5      case $removing\ handler\ T_{crem}$                /* resource release or task completion */
6          remove($T_{crem}$, $\sigma$) ;
7          $T_c \leftarrow T_c - T_{crem}$ ;
8      case $adding\ handler\ T_{cnew}$                  /* resource request or task arrival */
9          $\sigma_{copy} \leftarrow$ insertByEDF($T_{cnew}$, $\sigma$) ;
10         if feasible($\sigma_{copy}$) then
11             $T_c \leftarrow T_c \cup T_{cnew}$ ;
12             $T_{cnew}.LCT \leftarrow T_{cnew}.DL$- $T_{cnew}.ExecTime$ ;
13             $\sigma \leftarrow \sigma_{copy}$ ;
14         end
15     case $LCT$                                                     /* LCT timeout */
16         $T_{exe} \leftarrow$ headOf($\sigma$);
17         return $T_{exe}$
18
19 end
20 $T_{handler} \leftarrow$ headOf($\sigma$);
21 setLCT($T_{handler}.LCT$);
22 for $T_i \in T_r$ do
```

The computeVD function sums the values of a thread and its dependencies. The function then divides the sum of values by the sum of execution times for a thread and its dependencies. This yields the aggregate value density the system can expect from executing the thread and all threads upon which it depends. The getDep function returns the thread that holds the resource that T_i is requesting.

The insertByEDF function inserts a task and its dependencies into a deadline ordered list. Initially, the function makes a copy of the schedule and inserts the new thread, T_i, into the schedule copy. The dependency chain is then iterated through and each dependency's deadline is tightened before the dependency is added to the schedule copy. The function returns the copy of the schedule without making any changes to the original.

C. Metascheduler Threads

The Metascheduler framework used to implement the AUA algorithm enforces scheduling state consistency on all threads in the system. The AUA algorithm is not directly aware of the distributable thread abstraction, however the primitive blocking, pausing, and abort states are sufficient to construct the DT abstraction in middleware. As a consequence, AUA may be used to schedule local-only threads without incurring any overhead associated with DTs.

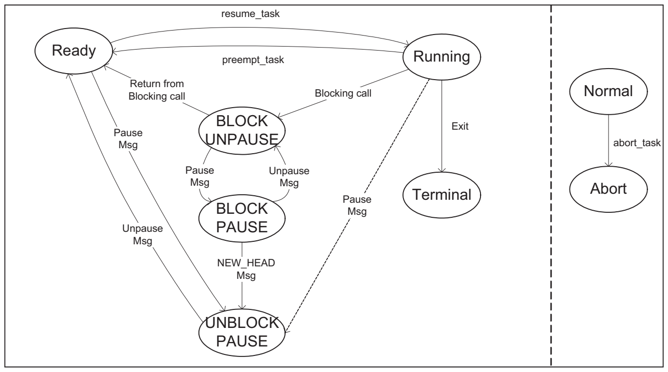


Fig. 4. Thread Scheduling States

In Figure 4, we present the various scheduling states supporting the distribution middleware. When a thread enters a PAUSE or BLOCK state, the scheduler is able to resolve resource contention and dependencies while respecting local mutual exclusion invariants. Furthermore, the PAUSE state is explicitly governed to allow coordinated control of all segments of a distributable thread.

IV. The *TP-TR* Protocol

A. Overview

The *TP-TR* TMAR protocol is an extension of the Alpha TMAR protocol described in [20]. The *TP-TR* protocol is instantiated in a software component called the Thread Integrity Manager (*TIM*). Every node which hosts distributable threads has a *TIM* component, which continually runs *TP-TR*'s three-phase polling operation.

The *TP-TR* specifies unique behaviors for nodes hosting the root segment of a DT. The *TIM* on each node is responsible for maintaining the health and coordinating any cleanup required for DTs rooted there. Downstream segments, then, manage their health by responding to health update information sent by the root. If health information fails to arrive for a given amount of time, the segment deems itself an *orphan* and commences autonomous cleanup. Once this occurs, the thread segment is effectively disconnected from the remainder of the thread's call-graph, and control is returned to application code in the context of an exceptional cleanup handler.

The operations of the *TIM* are considered to be administrative operations, and they are conducted with scheduling eligibility that exceeds all application threads. As a consequence, we ignore the (comparatively small, and bounded) processing delays on each node in the analysis below.

In the exposition below we provide informal observations of the protocol's timeliness properties. For clarity and brevity, we have not included the full proofs.

B. Thread Polling

In the first phase, the root node of a given distributed thread (DT) regularly broadcasts an ROOT_ANNOUNCE message to all nodes within the system. The ROOT_ANNOUNCE message is sent every T_p, or polling interval. Figure 5 illustrates the polling process for a healthy thread.

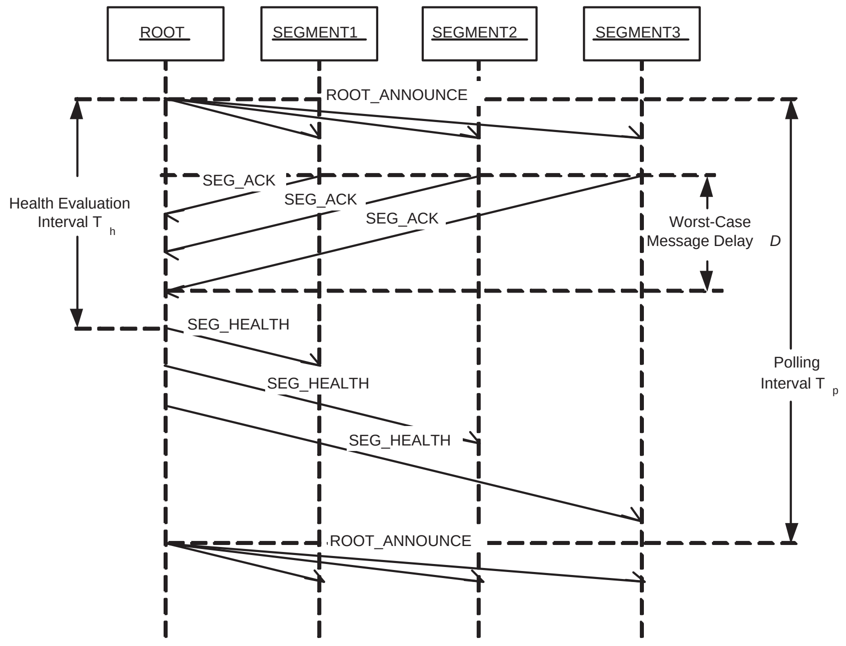


Fig. 5. *TP-TR* Operation — Healthy Thread

Observation TP-TR1: *Every healthy segment of a healthy thread will receive the ROOT_ANNOUNCE* *message at an interval not exceeding $T_p + D$. If the segment does not receive this message in that interval, either the root node has failed or the segment has become disconnected. The segment is thus orphaned.*

In the second phase, all nodes that are hosting segments of that given DT respond to the ROOT_ANNOUNCE with a segment acknowledgment (SEG_ACK) message.

Observation TP-TR2: *The root node will receive a SEG_ACK message from every healthy segment within a delay of $2D$ following a ROOT_ANNOUNCE broadcast. Thus, the thread health evaluation time T_h may be tuned as a function of the worst case message delay to ensure that no acknowledgment messages are missed. Furthermore, the worst case latency after which a broken thread will be detected is $2T_h$.*

In the last phase, the root node waits for the health evaluation interval T_h to expire before examining the information it has received from the SEG_ACK messages to determine the status of the DT (broken or unbroken). If the DT is determined to be unbroken, the root sends health update (SEG_HEALTH) messages to all segments of the DT, refreshing them. If there is a break in the DT, the root node refreshes only segments of the DT deemed healthy, and enters the recovery state to deal with the break.

Observation TP-TR3: *Every healthy segment of a healthy thread will have received a SEG_HEALTH message within $T_h + D$ of the receipt of any ROOT_ANNOUNCE message. Therefore, every healthy segment of a healthy thread will receive a SEG_HEALTH at a maximum interval of $T_p + T_h + D$. Segments may thus evaluate their health at a constant interval, irrespective of the dynamics of the system.*

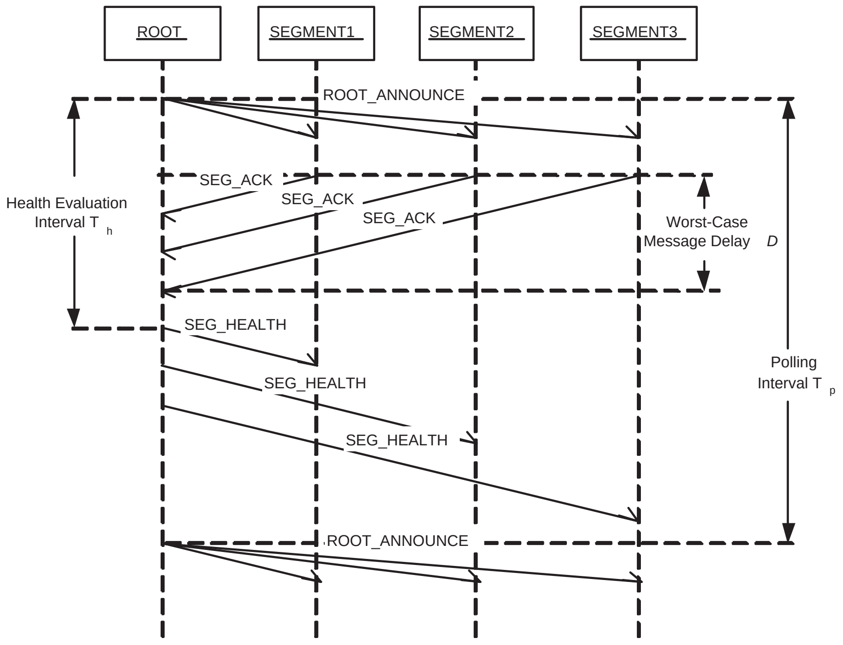


Fig. 6. *TP-TR* Operation — Unhealthy Thread Entering Recovery

C. *Recovery*

Recovery coordinated by *TP-TR* is considered to be an administrative function, and carries on below the level of application scheduling. While recovery proceeds, the thread-polling activities continue concurrently. This allows the protocol to recognize and deal with multiple simultaneous breaks, and even simultaneous cleanup operations.

Recovery from a thread break proceeds through four steps:

1) Pausing the thread and waiting for pause acknowledgment,
2) Determining which segment will be the new head,
3) Notifying the new head segment that it may continue to execute, and
4) Unpausing the distributed thread.

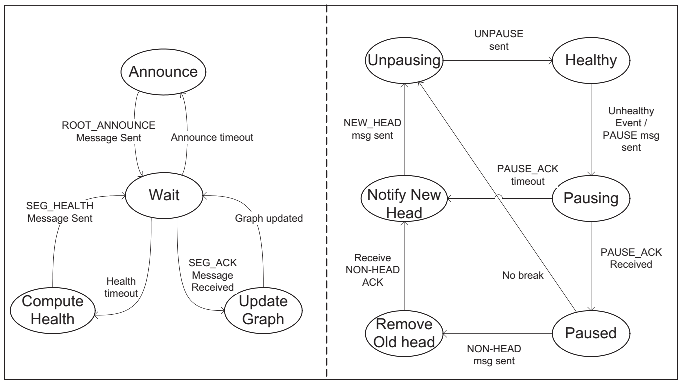


Fig. 7. High-level State Diagram – Root Segment

Figure 7 (on the right-hand side) illustrates the states experienced by an individual thread from the standpoint of its root segment. In the first step, the recovery operation broadcasts a PAUSE message and waits. The recovery thread continues waiting until it either receives a PAUSE_ACK message from the current head of the thread or a user-specified amount of time lapses without a PAUSE_ACK message being received. In the second step, the recovery operation analyzes the thread's distributed call-graph and finds the farthest contiguous thread segment from the root. This segment will be the new head. If the old head still exists after this step, the recovery thread must terminate the old head and wait for an acknowledgement that this action has been completed. In the third step, the recovery thread sends a NEW_HEAD message to the node hosting the new head. In the fourth step, the recovery thread broadcasts an UNPAUSE message to all nodes within the system. The recovery operation then terminates, and the thread is considered healthy.

Observation TP-TR4: *Based on Observation TP-TR3 above, the root node will identify a broken thread within $2T_h$, will pause the DT within $2D$, and will select and activate a new head within $2D$. Therefore, the worst case latency from detecting a failure and identifying a new head is $2T_h + 4D$.*

From here, the point of execution is return to application code at the new head at the point of remote invocation. An error code is returned to indicate that a thread integrity failure has occurred, and it is the responsibility of the application programmer to decide what should be done to proceed.

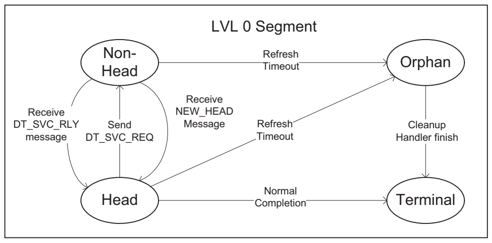


Fig. 8. High-level State Diagram – Segment

D. *Orphan Cleanup*

When a segment has not been refreshed for a specified amount of time it is flagged as an orphan and removed during orphan cleanup, which is performed periodically on all nodes within the system. Orphan cleanup is considered an administrative function, and occurs outside the context of application scheduling. The integrity manager determines which locally hosted segments, if any, are orphans. The manager then schedules the respective cleanup code to be run for each orphan. Orphan cleanup serves both to remove segments that follow a break in the DT (called *thread trimming*) and to remove the entirety of threads that have lost their root.

Observation TP-TR5: *If every segment of every thread is scheduling using the AUA scheduling discipline, their recovery times are bounded by the assured execution time in each of their abort handlers. By observation TP-TR3, every unhealthy segment will detect that it is an orphan and clean up within* $T_h + D + T_c$*, where* T_c *is the worst case completion time of the segment's cleanup handler.*

V. Implementation Experience

A. *Implementation in Tempus*

The *AUA* scheduling algorithm and *TP-TR* thread integrity protocol were implemented in a custom distributed middleware environment developed in Virginia Tech's Real-Time Systems Laboratory. This environment consists of *Tempus* [22], an implementation of the distributable threads abstraction in the C programming language. In addition, a pluggable scheduling framework called the *Metascheduler* [25] facilitates the composition of user-defined scheduling policies such as *AUA*.

B. *Experimental Setup*

The experiments presented below were performed on a small testbed of Intel Pentium III-based PC's running QNX Neutrino 6.2.1. The interconnect consists of commodity 10 megabit/sec interfaces on a switched Ethernet network. Each machine hosts an instance of the Tempus middleware and Metascheduler scheduling framework.

C. Single Node AUA Performance

A number of experiments were carried out to establish the behavior of the AUA scheduling approach in a single node context. We measured the Accrued Utility Ratio (AUR), Deadline Satisfaction Ratio (DSR), and Deadline Miss Load (DML) produced by our implementation under a variety of load and task structure conditions.

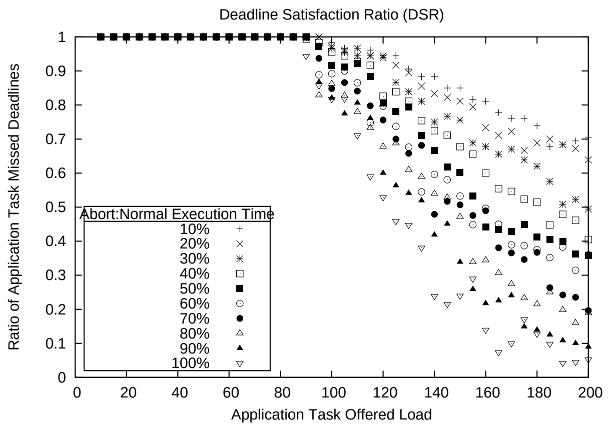


Fig. 9. Deadline Satisfaction Ratio

The DSR metric is convenient for comparison to traditional deadline-driven scheduling approaches. As with our *AUR* measurements, we conducted experiments to profile deadline satisfaction over a range of load conditions. On the horizontal axis, the offered application task load is ramped from zero to 200% of available CPU capacity. Up to a certain load — when the system is "underloaded" — every deadline is satisfied. As the load increases beyond the "deadline miss load" (presented in detail below), an increasing number of tasks fail to complete by their deadlines.

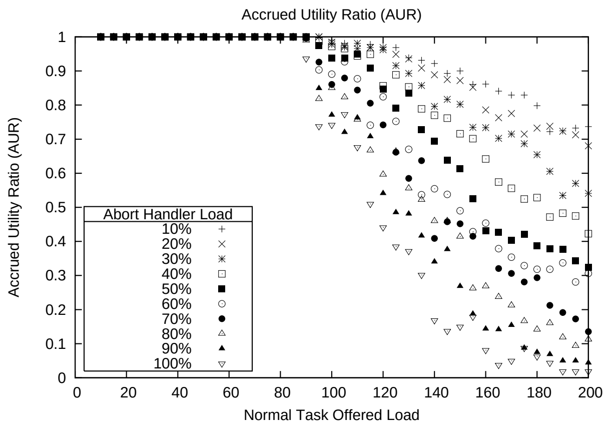


Fig. 10. Accrued Utility Ratio

The AUR is a direct measurement of the "value" delivered to the application tasks. The data presented in Figure 10 illustrates the accrued utility as the offered load on the scheduler is elevated between 0% and 200%. As we have argued above, AUA delivers a 1.0 accrued utility ratio — it satisfies the deadline of all tasks, irrespective of their utility — when operating in underload. This data bears out the claim that AUA is equivalent to DASA, and hence EDF, in underloads.

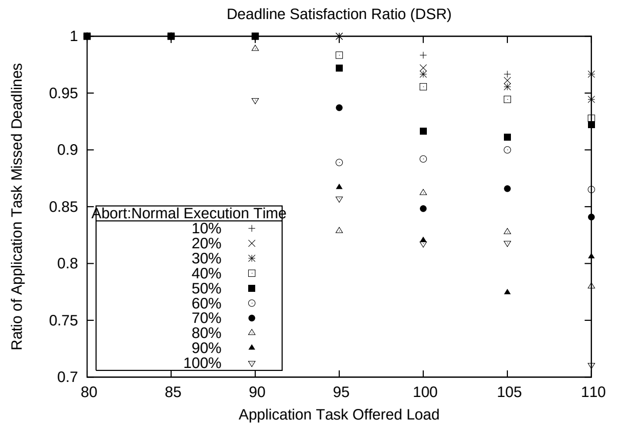


Fig. 11. Deadline Satisfaction Ratio (detail)

Upon closer inspection (see Figure 11), it can be seen that the highest load at which AUA misses no deadlines is a function of the currently accepted load of abort handlers. Intuitively, this is the correct behavior since AUA effectively reserves schedule to ensure that cleanup handlers are feasible in the presence of any offered application load. The data show that AUA is nevertheless able to degrade gracefully as the load increases, continuing to meet significant fractions of the time constraints despite operating in overload.

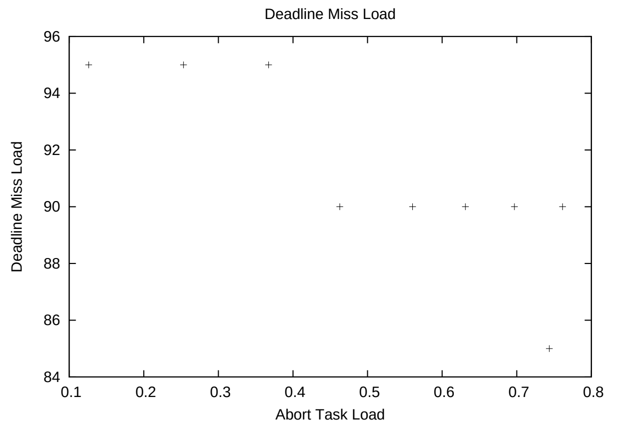


Fig. 12. Deadline Miss Load

The *DML* of a scheduler is defined to be the offered load under which the scheduler begins missing task deadlines. Ideally, the *DML* would occur at precisely 100% load; the scheduler would never miss a feasible deadline. Because of implementation-induced overhead such as context switch latency and time spent in scheduler and operating system code, it is not possible to achieve this theoretical maximum.

Furthermore, the overhead associated with scheduler and OS logic becomes more pronounced as task time constraints decrease, becoming very pronounced when the task execution times are on the same order as scheduling latencies. In addition, we show in Figure 12 that the *DML* is also adversely affected by the abort load induced by the currently-accepted set of threads. However, the algorithm performs reasonably well for low abort loads, missing no deadlines at 95% of theoretical capacity, despite a 30% load for abort reservations.

VI. Conclusions and Future Work

In this paper, we have presented a real-time scheduling algorithm AUA paired with a distributed thread integrity protocol called TP-TA. Together, the algorithm and the protocol schedule and provide thread integrity for distributed threads across a system in the Real-Time CORBA Case II model. In addition, we are able to provide bounds on the worst-case fault detection and cleanup time for threads experiencing partial failures.

The experimental results presented demonstrate the effectiveness of the *AUA* scheduling algorithm scheduling a variety of tasks loads induced by distributed threads in the *Tempus* middleware environment. Furthermore, we argue that this suite provides a useful framework for implementing resilient distributed computational activities in systems subject to partial (crash) failures.

Our work can be extended in several directions. Examples include considering mobile, ad-hoc networks, relaxing the upper bounds on communication delays, relaxing the requirements for reliable communication, and richer assurance semantics for abort handlers.

Acknowledgments

This work was supported by the U.S. Office of Naval Research under Grant N00014-99-1-0158 and N00014-00-1-0549, by the MITRE Corporation, and by QNX Software Systems Ltd. through a software grant.

References

[1] OMG, "Real-time corba 2.0: Dynamic scheduling specification," Object Management Group, Tech. Rep., September 2001, oMG Final Adopted Specification, http://www.omg.org/docs/ptc/01-08-34.pdf.

[2] E. D. Jensen, A. Wellings, R. Clark, and D. Wells, "The distributed real-time specification for java: A status report," in *Proceedings of The Embedded Systems Conference*, 2002.

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

[4] E. D. Jensen and J. D. Northcutt, "Alpha: A non-proprietary operating system for large, complex, distributed real-time systems," in *IEEE Workshop on Experimental Distributed Systems*, 1990, pp. 35–41.

[5] T. O. G. R. I. R.-T. Group, *MK7.3a Release Notes.* Cambridge, Massachusetts: The Open Group Research Institute, October 1998.

[6] GlobalSecurity.org, "Multi-platform radar technology insertion program," http://www.globalsecurity.org/intell/systems/mp-rtip.htm/.

[7] ——, "Multi-sensor command and control aircraft," http://www.globalsecurity.org/military/systems/aircraft/e-767-mc2a.htm.

[8] ——, "E-3 sentry (awacs)," http://www.globalsecurity.org/military/systems/aircraft/e-3.htm/.

[9] R. Clark, E. D. Jensen, *et al.*, "An adaptive, distributed airborne tracking system," in *IEEE Workshop on Parallel and Distributed Real-Time Systems*, ser. LNCS, vol. 1586. Springer-Verlag, April 1999, pp. 353–362.

[10] GlobalSecurity.org, "Bmc3i battle management, command, control, communications and intelligence," http://www.globalsecurity.org/space/systems/bmc3i.htm/.

[11] CCRP, "Network centric warfare," http://www.dodccrp.org/ncwPages/ncwPage.html.

[12] W. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quaterly*, vol. 21, pp. 177–185, 1974.

[13] E. D. Jensen, C. D. Locke, and H. Tokuda, "A time-driven scheduling model for real-time systems," in *IEEE Real-Time Systems Symposium*, Dec. 1985, pp. 112–122.

[14] B. Ravindran, E. D. Jensen, and P. Li, "On recent advances in time/utility function real-time scheduling and resource management," in *IEEE ISORC*, May 2005, pp. 55 – 60.

[15] C. D. Locke, "Best-effort decision making for real-time scheduling," Ph.D. dissertation, CMU, 1986, cMU-CS-86-134.

[16] R. K. Clark, "Scheduling dependent real-time activities," Ph.D. dissertation, CMU, 1990, cMU-CS-90-155.

[17] H. Wu, B. Ravindran, *et al.*, "Utility accrual scheduling under arbitrary time/utility functions and multiunit resource constraints," in *IEEE RTCSA*, August 2004.

[18] R. K. Clark, E. D. Jensen, and F. D. Reynolds, "An architectural overview of the alpha real-time distributed kernel," in *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, April 1992.

[19] J. D. Northcutt and R. K. Clark, "The alpha operating system: Programming model," Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Archons Project Technical Report 88021, February 1988.

[20] J. Goldberg, I. Greenberg, R. K. Clark, E. D. Jensen, K. Kim, and D. M. Wells, "Adaptive fault-resistant systems (chapter 5: Adpative distributed thread integrity)," Computer Science Laboratory, SRI International, Menlo Park, CA., Tech. Rep. csl-95-02, January 1995, http://www.csl.sri.com/papers/sri-csl-95-02/.

[21] S. Nagy and A. Bestavros, "Admission control for soft-transactions in accord," in *Third IEEE Real-Time Technology and Applications Symposium (RTAS'97*, 1997, p. 160.

[22] P. Li, B. Ravindran, H. Cho, and E. D. Jensen, "Scheduling distributable real-time threads in tempus middleware," in *IEEE Conference on Parallel and Distributed Systems*, July 2004, pp. 187 – 194.

[23] D. L. Mills, “Improved algorithms for synchronizing computer network clocks,” *IEEE/ACM Transactions on Networking*, vol. 3, pp. 245–254, June 1995.

[24] R. K. Clark, “Scheduling dependent real-time activities,” Ph.D. dissertation, Carnegie Mellon University, August 1990.

[25] 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, no. 9, pp. 613 – 629, Sept. 2004.