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

Edward Curley, Jonathan Anderson, Binoy Ravindran
ECE Department, Virginia Tech
Blacksburg, VA 24061, USA
{alias,andersoj,binoy}@vt.edu

E. D. Jensen
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., thread segments 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, and failure-exception notification must be delivered to the farthest, contiguous surviving thread segment for resuming thread execution. We present a real-time scheduling algorithm called AUA*, and 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, and maximize the total thread accrued timeliness utility. 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.*

1. 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) [19] and Sun's upcoming Distributed Real-Time Specification for Java (DRTSJ) standard [13] directly provides that as a first-class abstraction. Distributable threads first appeared in the Alpha OS [17, 12] and later in Alpha's descendant, the MK7.3 OS [20].

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.

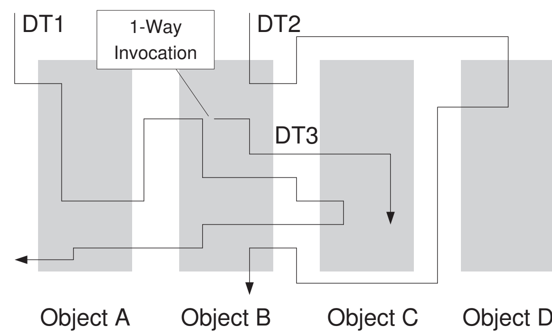

Figure 1. Distributable Threads

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 [19] shows the execution of threads.

The Real-Time CORBA specification envisions four distributed scheduling "cases", summarized in Table 1. 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

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 1. Real-Time CORBA Distributed Scheduling Cases [19, Section 3.8]

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 Cases 3 or 4.

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 need the strongest possible assurances on activity timeliness behavior that are feasible under the circumstances. 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 [8]), surveillance aircraft [9, 7, 2]), and network-centric warfare [6, 1]). These dynamic systems have traditionally been designed as traditional real-time systems, requiring worst-case load and failure models. Designers and users of these systems have found that, due to the long lifetimes of these systems, the increasingly dynamic execution environment, and the flexible way they are employed in real-world situations, the deterministic worst-case analysis performed at design and implementation time enforces unacceptable bounds on the use of the system.

1.1. 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 the thread's 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 urgency/low importance thread must cause minimal interference to high urgency/high importance threads. On the other hand, if orphans of a failed low urgency/low importance thread hold resources which are blocking high urgency/low importance threads, then the cleanup activity must have execution eligibility that reflects 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 detected and thread integrity is maintained through thread integrity protocols.

In this model, we explicitly assume that thread overruns are at best wasteful, and at worst degrade system safety. Thus, we demand that the scheduling algorithm ensure that overruns do not occur. Similarly, this model assumes that the application-specified abort code (if any) is unexceptionally the correct response to overrun or failure conditions. If the application wishes to schedule request new, complex, and less restricted handler, it is free to spawn such an activity as a separate thread.

We present a real-time scheduling algorithm called *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 are appropriate for the RTC2/DRTSJ distributable threads programming model. We specify (end-to-end) time constraints on threads using the time/utility function (or TUF) [11] timing model that generalizes the classical deadline time constraint, and which decouples thread urgency from thread importance.

[1] Under a continuation model, the orphan segments may be allowed to continue execution transparently, after cleanup has completed. A discussion of termination versus continuation (or "resumption") models is beyond the scope of this paper. See [5] for a complete treatment of this topic.

This decoupling facilitates thread scheduling that favors more important threads over less important ones, irrespective of their urgency, during overloads when all threads cannot be completed. Threads may be created at arbitrary times, and may span nodes that are subject to arbitrary crash failures. We consider a single-hop network model.

We show that AUA achieves optimal total accrued utility during the special case of underloads and no failures, and maximizes the total utility, as much as possible, during overloads and failures. We establish that AUA, in conjunction with TP-TR, bounds cleanup and recovery times, and thereby bounds thread starvation durations. We also implement AUA and TP-TR in an RTC2-like real-time middleware. Our experimental measurements from the implementation validate the algorithm/protocol properties and confirm their effectiveness.

Thread integrity protocols have been developed in the past—e.g., Alpha's Thread Polling protocol [4, 18], the Node Alive protocol [10], and their adaptive versions [10]. However, to the best of our knowledge, no thread integrity solution — i.e., a stand-alone protocol or one that is coupled with a scheduling algorithm — exists that provides 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 2, we discuss the models of our work and state the algorithm/protocol objectives. Section 3 presents the AUA algorithm, and Section 4 presents the TP-TR protocol. In Section 5, we discuss our implementation experience. We conclude the paper and identify future work in Section 6.

2. Models and Algorithm/Protocol Objectives

2.1. Distributable Thread Abstraction

Distributable 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. The first segment in the section results from an invocation from another node and the last segment in the section performs a remote invocation.

Threads may be created at any node at any time. Upon arrival at a node, threads are assumed to present execution time estimates of normal code and abort code (or exception handler code) for 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).

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

2.2. Timeliness Model

We specify the time constraint of each thread using a TUF. A TUF specifies the utility of completing a thread as a function of its completion time. Fig. 2 shows downward "step" shaped TUFs.

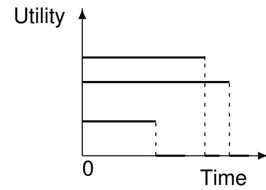

Figure 2. Example Step TUFs

A TUF decouples importance and urgency of a thread—i.e., urgency is measured as a deadline on the X-axis, and importance is denoted by utility on the Y-axis. This decoupling is a key property of TUFs, as a thread's urgency is typically orthogonal to its relative importance—e.g., the most urgent thread can be the least important, and vice versa; the most urgent can be the most important, and vice versa.

A thread T_i's TUF is denoted as $U_i\left(t\right)$. A classical deadline is unit-valued—i.e., $U_i(t) = \{0, 1\}$, since importance is not considered. Downward step TUFs (Fig. 2) are a generalization of classical deadlines where $U_i(t) = \{0, \{n\}\}$. We focus on downward step functions, and denote the maximum, constant utility of a TUF $U_i\left(\right)$, 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 critical time X_i, which, for a downward step TUF, is its discontinuity point. We assume that $U_i\left(t\right) > 0, \forall t \in [I_i, X_i]$ and $U_i\left(t\right) = 0, \forall t \notin [I_i, X_i], \forall i$.

If a thread's critical time is reached and its execution has not been completed, a failure-exception is raised, and exception handlers are released for aborting all partially executed thread sections (for releasing system resources). The handlers' time constraints are also specified using TUFs.

2.3. System and Failure Models

We consider a system model wherein 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 [19] 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 [12] and Tempus middleware [14].[2]

We consider a single hop network model (e.g., a LAN), where nodes are interconnected through a hub or 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 using a protocol such as [16]. 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.

2.4. Algorithm/Protocol Objective

Our objective is to 1) maximize the total thread accrued utility and 2) bound the orphan cleanup and recovery time. Note that maximizing the total utility subsumes meeting all TUF critical times as a special case. When all critical times are met (which is possible during underloads), the total utility is the optimum possible. During overloads, the goal is to maximize the total utility as much as possible, since not all critical times can be met.

The orphan cleanup and recovery time is the time from failure detection till the time of notifying the farthest, contiguous surviving thread segment (from where execution can be potentially resumed), after aborting all orphans.

[2]RTC2 also describes Cases 1, 3, and 4, which describe non real-time, global and multilevel distributed scheduling, respectively [19]. However, RTC2 does not support Cases 3 and 4.

3. The AUA Algorithm

3.1. Rationale

T_r	current set of accepted, unscheduled threads
T_c	current handlers accepted in the system
T_{rnew}	new thread arrival event
T_{cnew}	new handler arrival 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.Utility	potential utility U_i 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

Table 2. Variables in AUA

In order to attain bounded recovery time for distributable threads, it is necessary to have a scheduling algorithm which guarantees a bound on the time required by each *orphaned* 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 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.

Algorithm 1: AUA Event Handler

```
   Data: T_r, T_c, event
   Result: selected thread to execute, T_exe
1  σ ← ∅ ;
2  for T_i ∈ T_c do  σ ← insertByEDF(T_i, σ) ;
3  switch event do
4      case removing handler T_crem
5          remove(T_crem, σ) ;
6          T_c ← T_c − T_crem ;
7      case adding handler T_cnew
8          σ_copy ← insertByEDF(T_cnew, σ) ;
9          if feasible(σ_copy) then
10             T_c ← T_c ∪ T_cnew ;
11             T_cnew.LCT ← T_cnew.DL-
               T_cnew.ExecTime ;
12             σ ← σ_copy ;
13         end
14     case LCT_Timeout
15         T_exe ← headOf(σ);
16         return T_exe
17
18 end
```

The AUA scheduling algorithm is a hybrid approach, seeking to maximize the total (summed) utility for all tasks in the system, subject to the guarantee that execution of those blocks of code designated as cleanup handlers will always complete by their TUF critical time (or deadline). 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 4.

The engineering choice made by AUA to provide deterministically feasible abort handlers is motivated by a desire to enforce system safety. Other approaches to executing cleanup code include amortizing the cleanup code into other system operations or delaying cleanup of the affected resources until they are needed by other tasks. The approach described in the AUA algorithm chooses to avoid the nondeterministic delays implied by these other methods. The guarantee, however, comes at a not insignificant cost, requiring deterministic analysis of the abort handlers, and therefore possibly over-aggressive rejection of tasks.

AUA traces its lineage to the *Dependent Activity Scheduling Algorithm* (DASA) introduced by Clark [3], and is equivalent to DASA if no abort handlers are introduced.

Algorithm 2: AUA Algorithm

```
   Data: T_r, T_c
   Result: selected thread to execute, T_exe
1  T_handler ← headOf(σ);
2  setLCT(T_handler.LCT);
3  for T_i ∈ T_r do
4  |   T_i.DEP ← getDep(T_i);
5  |   T_i.UD ← computeUD(T_i);
6  end
7  σ_tmp ← sortByUD(T_r);
8  for T_i ∈ σ_tmp do
9  |   if not inSchedule(T_i, σ) then
10 |   |   σ_copy ← insertByEDF(T_i, σ);
11 |   |   if feasible(σ_copy) then σ ← σ_copy ;
12 |   end
13 end
14 T_exe ← headOf(σ);
15 return T_exe ;
```

3.2. Algorithm Overview

The AUA algorithm is presented in two parts, an event handler in Algorithm 1 and the core scheduling code in Algorithm 2. 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.

Operation	Description
reqResource(T_i)	returns the resource requested by T_i
owner(R)	returns the thread that is currently holding resource R
headOf(σ)	return the first thread in schedule σ
sortByUD(σ)	return a new schedule sorted by non-increasing utility density (UD)
insert(T, σ, i)	insert thread T into ordered list σ at position 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)	remove T from the ordered list σ if T is in σ
feasible(σ)	return 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)	set a timer to wake up the scheduler at time t
copySchedule(σ)	make a copy of schedule σ

Table 3. Operations used in AUA

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 utility accrual optimization step. Thread segments are inserted into this schedule using the DASA algorithm, considering thread segments in decreasing order of their utility 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.*

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 [14] 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 3. A high-level description of the AUA scheduling algorithm is presented in Algorithm 2, 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 2.

The setLCT function sets a timer that will wake the scheduler when the next LCT arrives. The dependencies and utility density (UD) of each thread are then calculated using getDep and computeUD, respectively. The function sortByUD then uses the utility density to create a list of all the threads in the system sorted in non-increasing order by UD. 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.

The computeUD function sums the utilities of a thread and its dependencies. The function then divides the sum of utilities by the sum of execution times for a thread and its dependencies. This yields the aggregate utility 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.

3.3. 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 abstraction in *Tempus* middleware. As a consequence, AUA may be used to schedule local-only threads without incurring any overhead associated with distributable threads.

In Figure 3, 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 thread.

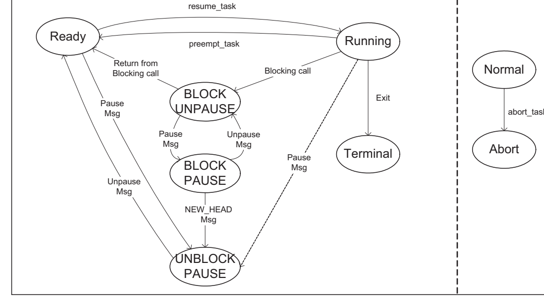

Figure 3. Thread Scheduling States

4. The TP-TR Protocol

4.1. Overview

The TP-TR TMAR protocol is an extension of the Alpha TMAR protocol described in [10]. 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 thread. The TIM on each node is responsible for maintaining the health and coordinating any cleanup required for threads 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.

4.2. Thread Polling

In the first phase, the root node of a given thread 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 4 illustrates the polling process for a healthy thread.

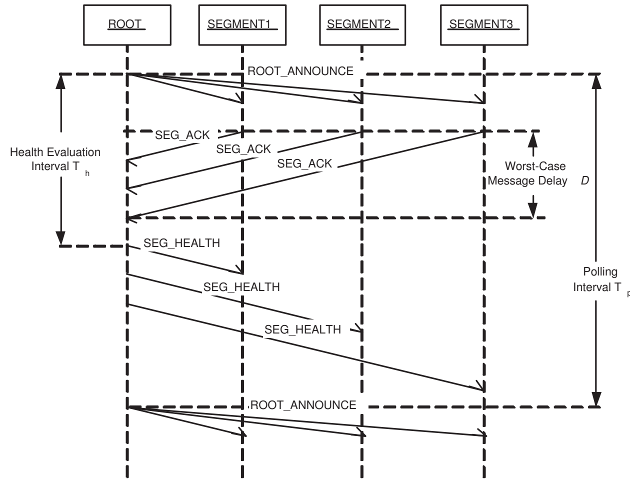

Figure 4. TP-TR **Operation — Healthy Thread**

Observation TPTR-1: *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 thread respond to the ROOT_ANNOUNCE with a segment acknowledgment (SEG_ACK) message.

Observation TPTR-2: *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 thread (broken or unbroken). If the thread is determined to be unbroken, the root sends health update (SEG_HEALTH) messages to all segments of the thread, refreshing them. If there is a break in the thread, the root node refreshes only segments of the thread deemed healthy, and enters the recovery state to deal with the break.

Observation TPTR-3: *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.*

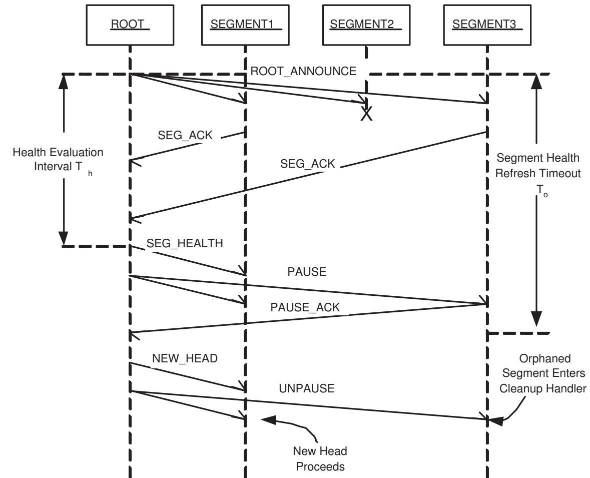

Figure 5. TP-TR **Operation — Unhealthy Thread Entering Recovery**

4.3. 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 thread.

Figure 6 (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

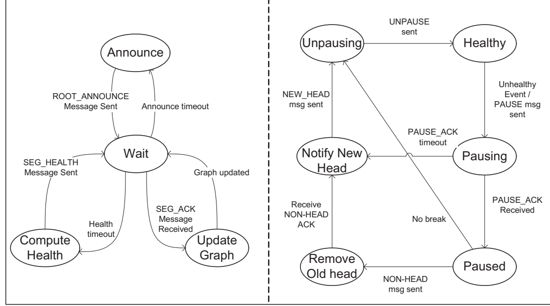

Figure 6. High-level State Diagram — Root Segment

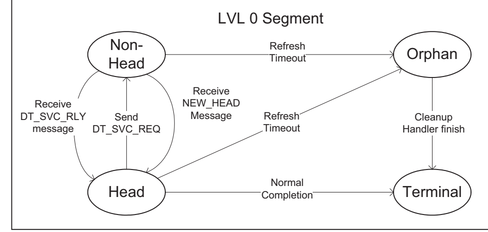

Figure 7. High-level State Diagram — Segment

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 TPTR-4: *Based on Observation TP-TR-3 above, the root node will identify a broken thread within $2T_h$, will pause the thread 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.

4.4. 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 distributable thread (called *thread trimming*) and to remove the entirety of threads that have lost their root.

Observation TPTR-5: *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-TR-3, 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.*

5. Implementation Experience

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* [14], an implementation of the distributable threads abstraction in the C programming language. In addition, a pluggable scheduling framework called the *Metascheduler* [15] facilitates the composition of user-defined scheduling policies such as AUA.

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.

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.

Deadline Satisfaction Ratio: The *DSR* metric is defined as the number of tasks which complete by their deadline divided by the total number of tasks. *DSR* is therefore 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 de-

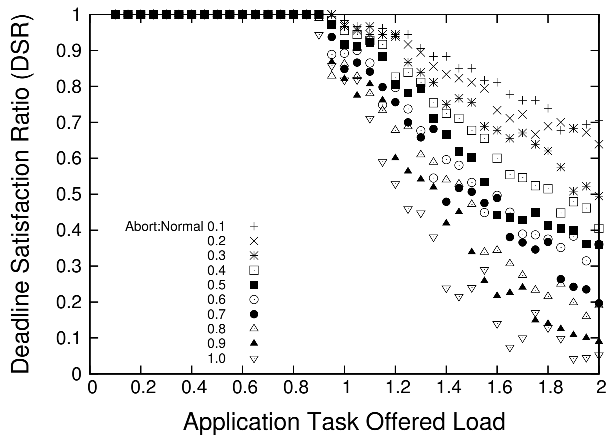

Figure 8. Deadline Satisfaction Ratio

tail below), an increasing number of tasks fail to complete by their deadlines.

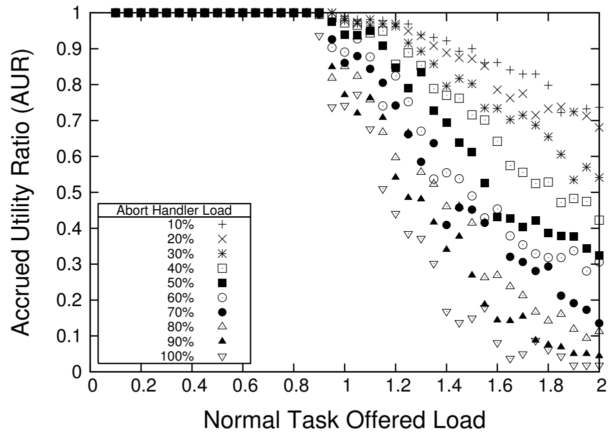

Figure 9. Accrued Utility Ratio

Accrued Utility Ratio: The *AUR* is defined as the ratio of utility earned by executing tasks successfully divided by the total utility of all tasks in the offered load. This is a direct measurement of the "value" delivered to the application tasks. The data presented in Figure 9 illustrates the accrued utility as the offered load on the scheduler is elevated from 0 to 2.0. 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.

Upon closer inspection (see Figure 10), 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

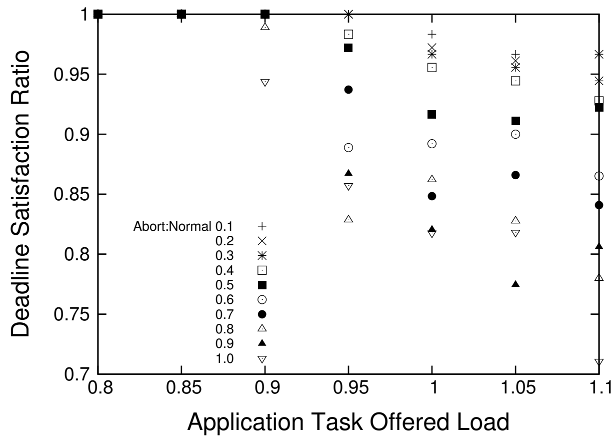

Figure 10. Deadline Satisfaction Ratio (detail)

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.

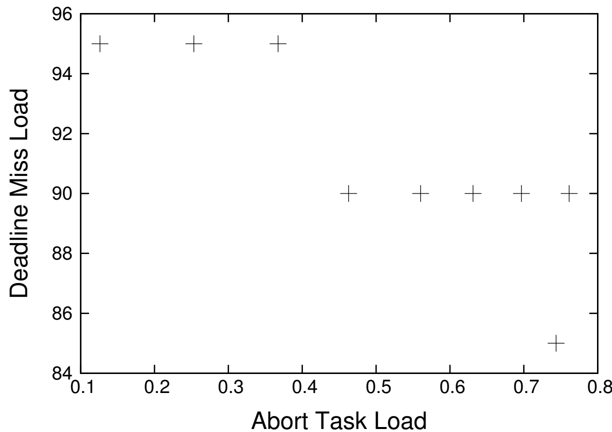

Figure 11. Deadline Miss Load

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 a load of 1.0; 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 11 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.

6. Conclusions and Future Work

In this paper, we have presented a real-time scheduling algorithm called AUA paired with a distributable thread integrity protocol called TP-TR. Together, the algorithm and the protocol schedule and provide thread integrity for threads across a system in the Real-Time CORBA Case II model. In addition, we 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 task loads induced by 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.

The approach presented in this paper provides assurances about the safety and consistency of the system by enforcing deterministic behavior for user-provided exception handlers. This approach is overly constraining for the desired class of systems, but represents a design point which may be used to understand a sufficient (but not minimally necessary) set of conditions for ensuring deterministic safety while providing graceful degradation in overloads.

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] CCRP. Network centric warfare. http://www.dodccrp.org/ncwPages/ncwPage.html.

[2] R. Clark, E. D. Jensen, et al. An adaptive, distributed airborne tracking system. In *IEEE Workshop on Parallel and Distributed Real-Time Systems*, volume 1586 of *LNCS*, pages 353–362. Springer-Verlag, April 1999.

[3] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, August 1990.

[4] 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.

[5] A. F. Garcia, C. M. Rubira, A. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197 – 222, November 2001.

[6] GlobalSecurity.org. BMC3I battle management, command, control, communications and intelligence. http://www.globalsecurity.org/space/systems/bmc3i.htm/.

[7] GlobalSecurity.org. E-3 sentry (AWACS). http://www.globalsecurity.org/military/systems/aircraft/e-3.htm/.

[8] GlobalSecurity.org. E-8 joint surveillance target attack radar system (jstars). http://www.globalsecurity.org/intell/systems/jstars.htm/.

[9] GlobalSecurity.org. Multi-sensor command and control aircraft. http://www.globalsecurity.org/military/systems/aircraft/e-767-mc2a.htm.

[10] 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). Technical Report csl-95-02, Computer Science Laboratory, SRI International, Menlo Park, CA., January 1995. http://www.csl.sri.com/papers/sri-csl-95-02/.

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

[12] 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*, pages 35–41, 1990.

[13] 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.

[14] 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*, pages 187 – 194, July 2004.

[15] 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*, 30(9):613 – 629, Sept. 2004.

[16] D. L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Trans. on Networking*, 3:245–254, June 1995.

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

[18] 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.

[19] OMG. Real-time CORBA 2.0: Dynamic scheduling specification. Technical report, Object Management Group, September 2001. OMG Final Adopted Specification, http://www.omg.org/docs/ptc/01-08-34.pdf.

[20] The Open Group Research Institute's Real-Time Group. *MK7.3a Release Notes*. The Open Group Research Institute, Cambridge, Massachusetts, October 1998.