# On Best-Effort Real-Time Assurances for Recovering from Distributable Thread Failures in Distributed Real-Time Systems

Binoy Ravindran$^\star$, Edward Curley$^\star$, Jonathan Anderson$^\star$, and E. Douglas Jensen$^\ddagger$

$^\star$ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{binoy,alias,andersoj}@vt.edu

$^\ddagger$The MITRE Corporation
Bedford, MA 01730, USA
jensen@mitre.org

**Abstract**

We consider the problem of recovering from failures of distributable threads in distributed real-time systems that operate under run-time uncertainties including those on thread execution times, thread arrivals, and node failure occurrences. When a thread encounters a node failure, it causes *orphans*. Under a termination model, the orphans must be detected and aborted, and exceptions must be delivered to farthest, contiguous surviving thread segment for resuming thread execution. Our application/scheduling model includes distributable threads and their exception handlers that are subject to time/utility function (TUF) time constraints and an utility accrual (UA) optimality criterion. A key underpinning of the TUF/UA scheduling paradigm is the notion of "best-effort" where high importance threads are always favored over low importance ones, irrespective of thread urgency. (This is in contrast to classical admission control models which favor feasible completion of admitted threads over admitting new ones, irrespective of thread importance.) We present a scheduling algorithm called HUA and a thread integrity protocol called TPR. We show that HUA and TPR bound the orphan cleanup and recovery time with bounded loss of the best-effort property. Our implementation experience of HUA/TPR using the emerging Reference Implementation of Sun's Distributed Real-Time Specification for Java demonstrates the algorithm/protocol's effectiveness.

## I. Introduction

Many distributed system applications (or portions of applications) are most naturally structured as a multiplicity of causally-dependent, flows of execution within and among objects, asynchronously and concurrently. The causal flow of execution can be a sequence—e.g., one that is caused by a series of nested, remote method invocations. It can also be caused by a series of chained, publication and subscription events, caused due to topical data dependencies—e.g., publication of topic A depends on subscription of topic B; B's publication, in turn, depends on subscription of topic C, and so on. Since partial failures are the common case rather than the exception in many distributed systems, applications often desire the causal, multi-node execution flow abstraction to exhibit application-specific, end-to-end integrity properties — one of the most important $raisons\ d'etre$ for building distributed systems. Real-time distributed applications also require end-to-end timeliness properties for the abstraction.

An abstraction for programming multi-node sequential behaviors and for enforcing end-to-end properties is *distributable threads* [1], [15]. Distributable threads first appeared in the Alpha OS [15], and later in MK7.3 [19]. They constitute the first-class programming and scheduling abstraction for multi-node sequential behaviors in Sun's emerging Distributed Real-Time Specification for Java (DRTSJ) [1]. In the rest of the paper, we will refer to distributable threads as *threads*, unless qualified.

A thread is a single logically distinct (i.e., having a globally unique ID) locus of control flow movement that extends and retracts through local and (potentially) remote objects. 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. The propagated thread context is intended to be used by node schedulers for resolving all node-local resource contention among threads such as that for node's physical (e.g., CPU) and logical (e.g., locks) resources, according to a discipline that provides acceptably optimal system-wide timeliness. Figure 1 shows the execution of three threads [16].

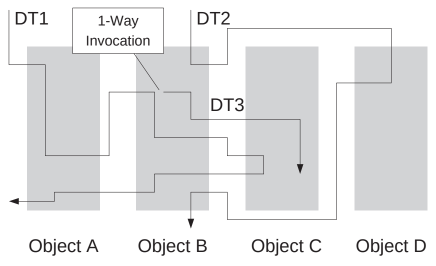

Fig. 1. Distributable Threads

We consider threads as the programming and scheduling abstraction in distributed real-time systems in the presence of uncertainties. These uncertainties include transient and sustained resource overloads (due to context-dependent thread execution times), arbitrary thread arrivals, and arbitrary node failures. Despite the uncertainties, such applications require the strongest possible assurances on thread timeliness behavior. Another distinguishing feature of motivating applications for this model (e.g., [3]) is their relatively long thread execution time magnitudes—e.g., milliseconds to minutes.

When overloads occur, meeting time constraints of all threads is impossible as the demand exceeds the supply. The urgency of a thread is sometimes orthogonal to the relative importance of the thread—-e.g., the most urgent thread may be the least important, and vice versa; the most urgent may be the most important, and vice versa. Hence when overloads occur, completing the most important threads irrespective of thread urgency is desirable. Thus, a distinction has to be made between urgency and importance during overloads. (During underloads, such a distinction generally need not be made, especially if all time constraints are deadlines, as optimal algorithms exist that can meet all deadlines—e.g., EDF [8].)

Deadlines cannot express both urgency and importance. Thus, we consider the *time/utility function* (or TUF) timeliness model [9] that specifies the utility of completing a thread as a function of that thread's completion time. We specify a deadline as a binary-valued, downward "step" shaped TUF; Figure 2(a) shows examples. A thread's TUF decouples its importance and urgency—urgency is measured on the X-axis, and importance is denoted (by utility) on the Y-axis.

Some real-time systems also have threads with *non-deadline* time constraints, such as those where the utility attained for thread completion *varies* (e.g., decreases, increases) with completion time. Figures 2(b)–2(c) show example TUFs from two defense applications [4], [13].

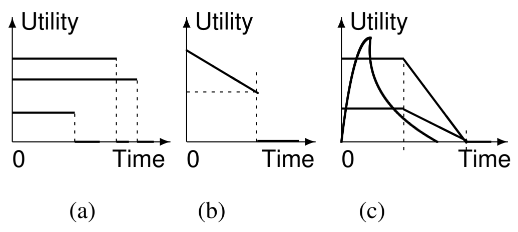

Fig. 2. Example TUF Time Constraints: (a) Step TUFs; (b) TUF of an AWACS [4]; and (c) TUFs of a Coastal Air defense System [13].

When thread time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued thread utility—e.g., maximizing the total thread accrued utility. Such criteria are called *utility accrual* (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that optimize UA criteria are called UA sequencing algorithms (see [17] for example algorithms).

UA algorithms that maximize total utility under downward step TUFs (e.g., [5], [12]) default to EDF during underloads, since EDF satisfies all deadlines during underloads. Consequently, they obtain the optimum total utility during underloads. During overloads, they inherently favor more important threads over less important ones (since more utility can be attained from the former), irrespective of thread urgency, and thus exhibit adaptive behavior and graceful timeliness degradation. This behavior of UA algorithms is called "best-effort" [12] in the sense that the algorithms strive their best to feasibly complete as many high importance threads — as specified by the application through TUFs — as possible.[1] Consequently, high importance threads that arrive at any time always have a very high likelihood for feasible completion (irrespective of their urgency). Note also that EDF's optimal timeliness behavior is a special-case of UA scheduling.

### A. Our Contributions: Time-Bounded Thread Cleanup with Bounded Loss of Best-Effort Property

When nodes transited by 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*. When threads encounter failures causing orphans, application-supplied exception handlers must be released for execution on the orphan nodes. Such handlers may have time constraints themselves and will compete for their nodes' processor along with other threads. Under a termination model, when handlers execute (not necessarily when they are released), they will abort the associated orphans after performing recovery actions that are necessary to avoid inconsistencies. Once all handlers complete their execution, the application may desire to resume the execution of the failed thread from the farthest, contiguous surviving thread segment from the thread's root. Such a coordinated set of recovery actions will preserve the abstraction of a continuous reliable thread.

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

Scheduling of the orphan-cleanup exception handlers along with threads must contribute to system-wide timeliness optimality. Untimely handler execution can degrade timeliness optimality—e.g.: high urgency handlers are delayed by low urgency non-failed threads, thereby delaying the resumption of high urgency failed threads; high urgency, non-failed threads are delayed by low urgency handlers.

A straightforward approach for scheduling handlers is to conceptually model them as traditional (single-node) threads, insert them into the ready queue when distributable threads arrive on nodes, and schedule them along with the threads on those nodes, according to a discipline that provides acceptable system-wide timeliness. This should be possible, as handlers are like single-node threads, with similar scheduling parameters (e.g., execution time, time constraints). However, constructing a schedule that includes a thread and its handler on a node implies that the thread *and* the handler will be dispatched for execution according to their order in the schedule. This is not true, as the handler needs to be dispatched only if and when the thread fails at an upstream node causing an orphan on the handler's node. Furthermore, when a thread is released for execution, which is a scheduling event, it is immediately ready for execution. However, its handler is released for execution only if and when the thread fails at an upstream node. Thus, constructing a schedule at a thread's release time on a node such that it also includes the thread's handler on the node will require a prediction of when the handler will be ready for execution in the future — a potentially impossible problem as there is no way to know if a thread will fail.

These problems can possibly be alleviated by considering a thread's failure time as a scheduling event and constructing schedules on the thread's orphan nodes that include the handlers at that time. However, this would mean that there is no way to know whether or not the handlers can feasibly complete until the thread fails. In fact, it is possible that when the thread fails, the schedulers on handlers' nodes' may discover that the handlers are infeasible due to node overloads — e.g., there are more threads on those nodes than can be feasibly scheduled, and there exists schedules of threads (on those nodes) excluding the handlers from which more utility can be attained than from ones including the handlers.

Another strategy that avoids this predicament and has been very often considered in the past (e.g., [2], [6], [18]) is classical *admission control*: When a thread arrives on a node, check whether a feasible schedule can be constructed on that node that includes all the previously admitted threads and their handlers, besides the newly arrived one and its handler. If so, admit the thread and its handler; otherwise, reject. But this will cause the very fundamental problem that is solved by UA schedulers through its best-effort decision making—i.e., a newly arriving thread is rejected because it is infeasible, despite that thread being the most important. In contrast, UA schedulers will feasibly complete the high importance newly arriving thread (with high likelihood), at the expense of not completing some previously arrived ones, since they are now less important than the newly arrived.

Thus, scheduling handlers with assured timeliness in dynamic distributed real-time systems involves an apparently paradoxical situation: a thread may arrive at any unknown time; in the event of its failure, which is unknown until the failure, handlers must be immediately released on all the thread orphan nodes, and as strong assurances as possible must be provided for the handlers' feasible completion.

We precisely address this problem in this paper. We consider distributable threads that are subject to TUF time constraints. Threads may have arbitrary arrival behaviors, may exhibit unbounded execution time behaviors (causing node overloads), and may span nodes that are subject to arbitrary crash failures. For such a model, we consider the scheduling objective of maximizing the total thread accrued utility.

We present a UA scheduling algorithm called *Handler-assured Utility Accrual scheduling algorithm* (or HUA) for thread scheduling, and a protocol called *Thread Polling with bounded Recovery* (or TPR) for ensuring thread integrity. We show that HUA in conjunction with TPR ensures that handlers of threads that encounter failures during their execution will complete within a bounded time, yielding bounded thread cleanup time. Yet, the algorithm/protocol retains the fundamental best-effort property of UA algorithms with bounded loss—i.e., a high importance thread that may arrive at any time has a very high likelihood for feasible completion. Our implementation experience of HUA/TPR using the emerging Reference Implementation of DRTSJ demonstrates the algorithm/protocol's effectiveness.

Similar to UA algorithms, integrity protocols for threads have been developed in the past—e.g., Alpha's Thread Polling protocol [15], the Node Alive protocol [7], and adaptive versions of Node Alive [7]. However, none of these protocols in conjunction with a scheduling algorithm provide time-bounded thread cleanup. Our work builds upon our prior work in [6] that provides bounded thread cleanup. However, [6] does so through admission control and thus suffers from unbounded loss of the best-effort property (we show this in Section III-C). In contrast, HUA/TPR provides bounded thread cleanup with bounded loss of the best-effort property. Thus, the paper's contribution is the HUA/TPR algorithm/protocol.

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 HUA and Section IV presents TPR. In Section V, we discuss our implementation experience. We conclude the paper in Section VI.

## II. Models and Algorithm/Protocol Objectives

### A. 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 [1], [15], [16].

Execution time estimates of the sections of a thread are assumed to be known when the thread arrives at the respective nodes. 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 total number of sections of a thread is assumed to be unknown a-priori, as a thread is assumed to make remote invocations and returns based on context-dependent application logic.

The application is thus comprised of a set of threads, denoted $\mathbf{T} = \{T_1, T_2, T_3, \ldots\}$.

### *B. Timeliness Model*

We specify the time constraint of each thread using a TUF. The TUF of a thread $T_i$ is denoted as $U_i\left(t\right)$. 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 *non-increasing* unimodal TUFs, as they encompass the majority of the time constraints of interest to us. Figures 2(a), 2(b), and two TUFs in Figure 2(c) show examples. (Unimodal TUFs are those for which any decrease in utility cannot be followed by an increase.)

Each TUF $U_i$ has an initial time $I_i$, which is the earliest time for which the function is defined, and a termination time $X_i$, which denotes the last point that the function crosses the X-axis. We assume that the initial time is the thread release time; thus a thread's absolute and relative termination times are the same. We also assume that $U_i\left(t\right) > 0, \forall t \in \left[I_i, X_i\right]$ and $U_i\left(t\right) = 0, \forall t \notin \left[I_i, X_i\right], i \in [1, n]$.

### *C. Exceptions and Abort Model*

Each section of a thread has an associated exception handler. We consider a termination model for thread failures including time-constraint violations and node failures.

If a thread has not completed by its termination time, a time constraint-violation exception is raised, and handlers are released on all nodes that host the thread's sections. When a handler executes (not necessarily when it is released), it will abort the associated section after performing compensations and recovery actions that are necessary to avoid inconsistencies—e.g., rolling back/forward, or making other compensations to logical and physical resources that are held by the section to safe states.

We consider a similar abort model for node failures. When a thread encounters a node failure causing orphans, TPR delivers failure-exception notifications to all orphan nodes of the thread. Those nodes then respond by releasing handlers which abort the orphans after executing compensating actions.

Each handler may have a time constraint, which is specified using a TUF. A handler's TUF's initial time is the time of failure of the handler's thread. The handler's TUF's termination time is relative to its initial time. Thus, a handler's absolute and relative termination times are *not* the same.

Each handler also has an execution time estimate. This estimate along with the handler's TUF are described by the handler's thread when the thread arrives at a node. Violation of the termination time of a handler's TUF will cause the immediate execution of system recovery code on that node, which will recover the thread section's held resources and return the system to a consistent and safe state.

### *D. 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 sections. The order of executing sections on a node is determined by the scheduler residing at the node. We consider the *Case 2* approach of Real-Time CORBA 1.2 [16] (which also supports distributable threads) for thread scheduling. According to this approach, node schedulers use the propagated thread scheduling parameters and independently schedule thread sections on their respective nodes to optimize the system-wide timeliness optimality criterion. Though this results in approximate, global, system-wide timeliness, Real-Time CORBA 1.2 supports the approach, due to its simplicity and capability for coherent end-to-end scheduling.

We consider a single hop network model (e.g., a LAN), with nodes interconnected through a hub or a switch. We assume a reliable message transport protocol with a worst case message delivery latency $D$.

We denote the set of nodes as $N_i \in N, i \in [1, m]$. We assume that all node clocks are synchronized using a protocol such as [14]. We consider an arbitrary, crash failure model for the nodes.

### *E. Scheduling Objectives*

Our primary objective is to maximize the total utility accrued by all the threads as much as possible. Further, the orphan cleanup and recovery time must be bounded. This is the time between the detection of a thread failure and the time of notifying the farthest, contiguous surviving thread segment (from where execution can be resumed), after aborting all the orphans of the thread. Moreover, the algorithm must exhibit the best-effort property of UA algorithms (described in Section I) to the extent possible.

## III. The HUA Algorithm

### A. Rationale

*Section Scheduling.* Since the task model is dynamic—i.e., when threads will arrive at nodes, and how many sections a thread will have are statically unknown, future scheduling events[2] such as new thread arrivals cannot be considered at a scheduling event. Thus, section schedules must be constructed on the system nodes solely exploiting the current system knowledge. Since the primary scheduling objective is to maximize the total thread accrued utility, a reasonable heuristic is a "greedy" strategy at each node: Favor "high return" thread sections over low return ones, and complete as many of them as possible before thread termination times, as early as possible (since TUFs are non-increasing).

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) originally introduced in [5]. 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.

However, a section may encounter failures. We first define the concept of a *section failure*:

*Definition 1 (Section Failure):* Consider a section $S_i$ of a distributable thread $T_i$. We say that $S_i$ has failed when (a) $S_i$ violates the termination time of $T_i$ *while* executing, thereby raising a time constraint-violation exception on $S_i$'s node; or (b) a failure-exception notification is received at $S_i$'s node regarding the failure of a section of $T_i$ that is upstream or downstream of $S_i$.

For convenience, we also define the concept of a *released handler*:

*Definition 2 (Released Handler):* A handler is said to be released for execution when its section fails according to Definition 1.

Since a section's best-case failure scenario is the absence of a failure for the section, the corresponding section PUD can be obtained as the utility accrued by executing the section divided by the time spent for executing the section. The section PUD for the worst-case failure scenario (one where the section fails, per Definition 1) can be obtained as the utility accrued by executing the handler of the section divided by the total time spent for executing the section and the handler.[3] The section's PUD can now be measured as the minimum of these two PUDs, as that represents the worst-case.

Thus, on each node, HUA examines thread sections 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 and its handler can be feasibly completed (we discuss section and handler feasibility later

[2]A "scheduling event" is an event that invokes the scheduling algorithm at a node.

[3]Note that, in the worst-case failure scenario, utility is accrued only for executing the section's handler; no utility is gained for executing the section, though execution time is spent for executing the section and its handler.

in this subsection). If infeasible, the section and its handler are rejected. The process is repeated until all sections are examined, and the schedule's first section is dispatched for execution on the node.

A section $S_i$ that is rejected can be the head of $S_i$'s thread $T_i$; if so, $S_i$ is reconsidered for scheduling at subsequent scheduling events on $S_i$'s node, say $N_i$, until $T_i$'s termination time expires.

If a rejected section $S_i$ is not a head, then $S_i$'s rejection is conceptually equivalent to the (crash) failure of $N_i$. This is because, $S_i$'s thread $T_i$ has made a downstream invocation after arriving at $N_i$ and is yet to return from that invocation (that's why $S_i$ is still a scheduling entity on $N_i$). If $T_i$ had made a downstream invocation, then $S_i$ had executed before, and hence was feasible and had a feasible handler at that time. $S_i$'s rejection now invalidates that previous feasibility. Thus, $S_i$ must be reported as failed and a thread break for $T_i$ at $N_i$ must be reported to have occurred to ensure system-wide consistency on thread feasibility. The algorithm does this by interacting with the TPR protocol.

This process ensures that the sections that are included in a node's schedule at any given time have feasible handlers. Further, all the upstream sections of their distributable threads also have feasible handlers on their respective nodes. Consequently, when any such section fails (per Definition 1), its handler and the handlers of all its upstream sections are assured to complete within a bounded time.

Note that no such assurances are afforded to sections that fail otherwise—i.e., the termination time expires for a section $S_i$, which has not completed its execution and is not executing when the expiration occurs. Since $S_i$ was not executing when the termination time expired, $S_i$ and its handler are not part of the feasible schedule at the expiration time. For this case, $S_i$'s handler is executed in a best-effort manner—i.e., in accordance with its potential contribution to the total utility (at the expiration time).

*Feasibility.* Feasibility of a section on a node can be tested by verifying whether the section can be completed on the node before the section's distributable thread's end-to-end termination time. Using a thread's end-to-end termination time for verifying the feasibility of a section of the thread may potentially overestimate the section's slack, especially if there are a significant number of sections that follow it in the thread. However, this is a reasonable choice, since we do not know the total number of sections of a thread. If the total number of sections of a thread is known a-priori, then better schemes (e.g., [10]) that distribute the thread's total slack (equally, proportionally) among all its sections can be considered.

For a section's handler, feasibility means whether it can complete before its *absolute* termination time, which is the time of thread failure plus the relative termination time of the section's handler. Since the thread failure time is impossible to predict, a reasonable choice for the handler's absolute termination time is the thread's end-to-end termination time plus the handler's termination time, as that will delay the handler's latest start time as much as possible. Delaying a handler's start time on a node is appropriate toward maximizing the total utility, as it potentially allows threads that may arrive later on the node but with an earlier termination time than that of the handler to be feasibly scheduled.

There is always the possibility that a new section $S_i$ is released on a node after the failure of another section $S_j$ at the node (per Definition 1) and before the completion of $S_j$'s handler on the node. As per the best-effort philosophy, $S_i$ must immediately be afforded the opportunity for feasible execution on the node, in accordance with its potential contribution to the total utility. However, it is possible that a schedule that includes $S_i$ on the node may not include $S_j$'s handler. Since $S_j$'s handler cannot be rejected now, as that will violate the commitment previously made to $S_j$, the only option left is to not consider $S_i$ for execution until $S_j$'s handler completes, consequently degrading the algorithm's best-effort property. In Section III-C, we quantify this loss.

## *B. Algorithm Overview*

HUA's scheduling events at a node include the arrival of a thread at the node, completion of a thread section or a section handler at the node, and the expiration of a TUF termination time at the node. To describe HUA, we define the following variables and auxiliary functions (at a node):

- $\mathcal{S}_r$ is the current set of unscheduled sections including a newly arrived section (if any). $S_i \in \mathcal{S}_r$ is a section. $S_i^h$ denotes $S_i$'s handler. $T_i$ denotes the thread to which a section $S_i$ and $S_i^h$ belong.
- $\sigma_r$ is the schedule (ordered list) constructed at the previous scheduling event. $\sigma$ is the new schedule.
- $U_i(t)$ denotes $S_i$'s TUF, which is the same as that of $T_i$'s TUF. $U_i^h(t)$ denotes $S_i^h$'s TUF.
- $S_i.X$ is $S_i$'s termination time, which is the same as that of $T_i$'s termination time. $S_i.ExecTime$ is $S_i$'s estimated remaining execution time.
- $H$ is the set of handlers that are released for execution on the node (per Definition 2), ordered by non-decreasing handler termination times. $H = \emptyset$ if all released handlers have completed.
- Function `updateReleaseHandlerSet()` inserts a handler $S_i^h$ into $H$ if the scheduler is invoked due to $S_i^h$'s release; deletes a handler $S_i^h$ from $H$ if the scheduler is invoked due to $S_i^h$'s completion. Insertion of $S_i^h$ into $H$ is at the position corresponding to $S_i^h$'s termination time.
- `notifyTPR(`$S_i$`)` declares $S_i$ as failed (by not sending a SEG_ACK message for $S_i$ in response to the ROOT_ANNOUNCE broadcast message of the TPR protocol — see Section IV for TPR details).
- `IsHead(`$S$`)` returns true if $S$ is a head; false otherwise.
- `headOf(`$\sigma$`)` returns the first section in $\sigma$.
- `sortByPUD(`$\sigma$`)` 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 $ExecTime$ will appear before any others with the same PUD.
- `Insert(`$S$`,`$\sigma$`,`$I$`)` inserts section $S$ in the ordered list $\sigma$ at the position indicated by index $I$; if entries in $\sigma$ exists with the index $I$, $S$ is inserted before them. After insertion, $S$'s index in $\sigma$ is $I$.

- `Remove(`$S,\sigma,I$`)` removes section $S$ from ordered list $\sigma$ at the position indicated by index $I$; if $S$ is not present at the position in $\sigma$, the function takes no action.
- `feasible(`$\sigma$`)` returns a boolean value indicating schedule $\sigma$'s feasibility. $\sigma$ is feasible, if the predicted completion time of each section $S$ in $\sigma$, denoted $S.C$, does not exceed $S$'s termination time. $S.C$ is the time at which the scheduler is invoked plus the sum of the $ExecTime$'s of all sections that occur before $S$ in $\sigma$ and $S.ExecTime$.

```
1:  input: $\mathcal{S}_r$, $\sigma_r$, $H$; output: selected thread $S_{exe}$;
2:  Initialization: $t := t_{cur}$; $\sigma := \emptyset$; $HandlerIsMissed :=$ false;
3:  updateReleaseHandlerSet ();
4:  for each section $S_i \in \mathcal{S}_r$ do
5:      if feasible($S_i$) =false then
6:          reject($S_i$);
7:      else $S_i.PUD = \min\left(\frac{U_i(t+S_i.ExecTime)}{S_i.ExecTime}, \frac{U_i^h(t+S_i.ExecTime+S_i^h.ExecTime)}{S_i.ExecTime+S_i^h.ExecTime}\right)$ ;
8:  $\sigma_{tmp}$ :=sortByPUD($\mathcal{S}_r$);
9:  for each section $S_i \in \sigma_{tmp}$ from head to tail do
10:     if $S_i.PUD > 0$ then
11:         Insert($S_i$, $\sigma$, $S_i.X$);
12:         Insert($S_i^h$, $\sigma$, $S_i.X + S_i^h.X$);
13:         if feasible($\sigma$) =false then
                Remove($S_i$, $\sigma$, $S_i.X$);
14:             Remove($S_i^h$, $\sigma$, $S_i.X + S_i^h.X$);
15:             if IsHead($S_i$) =false and $S_i \in \sigma_r$ then
16:                 notifyTPR($S_i$);
17:     else break;
18: if $H \neq \emptyset$ then
19:     for each section $S^h \in H$ do
20:         if $S^h \notin \sigma$ then
21:             $HandlerIsMissed :=$ true;
22:             break;
23: if $HandlerIsMissed :=$ true then
24:     $S_{exe}$ :=headOf($H$);
    else
25:     $\sigma_r := \sigma$;
26:     $T_{exe}$:=headOf($\sigma$);
27: return $S_{exe}$;
```

**Algorithm 1**: HUA: High Level Description

Algorithm 1 describes HUA at a high level of abstraction. When invoked at time $t_{cur}$, HUA first updates the set $H$ (line 3) and checks the feasibility of the sections. If a section's earliest predicted completion time exceeds its termination time, it is rejected (line 6). Otherwise, HUA calculates the section's PUD (line 7). To compute a section's PUD, HUA determines the PUDs for the best-case and worst-case failure scenarios and determines the minimum of the two.

The sections are then sorted by their PUDs (line 8). In each step of the *for*-loop from line 9 to line 17, the section with the largest PUD and its handler are inserted into $\sigma$, if it can produce a positive PUD. The schedule $\sigma$ is maintained in the non-decreasing order of section termination times. Thus, a section $S_i$ is inserted into $\sigma$ at a position that corresponds to $S_i.X$ in $\sigma$'s non-decreasing termination time order. $S_i^h$ is similarly inserted into $\sigma$ at a position corresponding to $S_i.X + S_i^h.X$.

After inserting a section $S_i$ and its handler $S_i^h$, the schedule $\sigma$ is tested for feasibility. If $\sigma$ becomes infeasible, then $S_i$ and $S_i^h$ are removed from $\sigma$ (lines 13–14). If a section $S_i$ that is removed from $\sigma$ is not a head and belonged to the schedule constructed at the previous scheduling event, then the TPR protocol is notified regarding $S_i$'s failure (lines 15–16).

If one or more handlers have been released but have not completed their execution (i.e., $H \neq \emptyset$; line 18), the algorithm checks whether any of those handlers are missing in the schedule $\sigma$ (lines 19–22). If any handler is missing, the handler at the head of $H$ is selected for execution (line 24). If all handlers in $H$ have been included in $\sigma$, the section at the head of $\sigma$ is selected (line 26).

*Asymptotic Complexity.* With $n$ sections, HUA's asymptotic cost is $O(n^2)$ (for brevity, we skip the analysis). Though this cost is higher than that of many traditional real-time scheduling algorithms, it is justified for applications with longer execution time magnitudes such as those that we focus on here. (Of course, this high cost cannot be justified for every application.)

## *C. Algorithm Properties*

We first describe HUA's bounded-time completion property for exception handlers:

*Theorem 1:* If a section $S_i$ fails (per Definition 1), then under HUA with zero overhead, its handler $S_i^h$ will complete no later than $S_i.X + S_i^h.X$ (barring $S_i^h$'s failure).

*Proof:* If $S_i$ violates the thread termination time at a time $t$ while executing, then $S_i$ was included in HUA's schedule constructed at the scheduling event that occurred nearest to $t$, say at $t'$, since only threads in the schedule are executed. Thus, both $S_i$ and $S_i^h$ were feasible at $t'$, and $S_i^h$ was scheduled to complete no later than $S_i.X + S_i^h.X$. Similar argument holds for the other cases:

If $S_i$ receives a notification on the failure of an upstream section $\bar{S}_i$ at a time $t$, then all sections from $\bar{S}_i$ to $S_i$ and their handlers are feasible on their respective nodes, as otherwise the thread execution would not have progressed to $S_i$ (and beyond if any). Thus, $S_i^h$ is scheduled to complete by $S_i.X + S_i^h.X$.

If $S_i$ receives a notification on the failure of a downstream section $\bar{S}_i$ at a time $t$, then all sections from $S_i$ to $\bar{S}_i$ and their handlers are feasible on their respective nodes, as otherwise the thread execution would not have progressed to $\bar{S}_i$. Thus, $S_i^h$ is scheduled to complete no later than $S_i.X + S_i^h.X$. ■

Consider a thread $T_i$ that arrives at a node and releases a section $S_i$ after the handler of a section $S_j$ has been released on the node (per Definition 2) and before that handler ($S_j^h$) completes. Now, HUA *may* exclude $S_i$ from a schedule until $S_j^h$ completes, resulting in some loss of the best-effort property. To quantify this loss, we define the concept of a *Non Best-effort time Interval* (or NBI):

*Definition 3:* Consider a scheduling algorithm $\mathcal{A}$. Let a section $S_i$ arrive at a time $t$ with the following properties: (a) $S_i$ and its handler together with all sections in $\mathcal{A}$'s schedule at time $t$ are not feasible at $t$, but $S_i$ and its handler are feasible just by themselves;[4] (b) One or more handlers (which were released before $t$) have not completed their execution at $t$; and (c) $S_i$ has the highest PUD among all sections in $\mathcal{A}$'s schedule at time $t$. Now, $\mathcal{A}$'s NBI, denoted $NBI_{\mathcal{A}}$, is defined as the duration of time that $S_i$ will have to wait after $t$, before it is included in $\mathcal{A}$'s feasible schedule. Thus, $S_i$ is assumed to be feasible together with its handler at $t + NBI_{\mathcal{A}}$.

We now describe the NBI of HUA and other UA algorithms including DASA [5], LBESA [12], and AUA [6] (under zero overhead):

*Theorem 2:* HUA's worst-case NBI is $t+\max_{\forall S_j \in \sigma_t}\left(S_j.X + S_j^h.X\right)$, where $\sigma_t$ denotes HUA's schedule at time $t$. DASA's and LBESA's worst-case NBI is zero; AUA's is $+\infty$.

*Proof:* The time $t$ that will result in the worst-case NBI for HUA is when $\sigma_t = H \neq \emptyset$. By NBI's definition, $S_i$ has the highest PUD and is feasible. Thus, $S_i$ will be included in the feasible schedule $\sigma$, resulting in the rejection of some handlers in $H$. Consequently, the algorithm will discard $\sigma$ and will select the first handler in $H$ for execution. In the worst-case, this process repeats for each of the scheduling events that occur until all the handlers in $\sigma_t$ complete (i.e., at handler completion times), as $S_i$ and its handler may be infeasible with the remaining handlers in $\sigma_t$ at each of those events. Since each handler in $\sigma_t$ is scheduled to complete by $\max_{\forall S_j \in \sigma_t}\left(S_j.X + S_j^h.X\right)$, the earliest time that $S_i$ becomes feasible is $t + \max_{\forall S_j \in \sigma_t}\left(S_j.X + S_j^h.X\right)$.

DASA and LBESA will examine $S_i$ at $t$, since a task arrival is always a scheduling event for them. Further, since $S_i$ has the highest PUD and is feasible, they will include $S_i$ in their feasible schedules at $t$ (before including any other tasks), yielding a zero worst-case NBI.

AUA will examine $S_i$ at $t$, since a task arrival at any time is also a scheduling event under it. However, AUA is a TUF/UA algorithm in the classical admission control mould and will reject $S_i$ in favor of previously admitted tasks, yielding a worst-case NBI of $+\infty$. ■

*Theorem 3:* The best-case NBI of HUA, DASA, and LBESA is zero; AUA's is $+\infty$.

*Proof:* HUA's best-case NBI occurs when $S_i$ arrives at $t$ and the algorithm includes $S_i$ and all handlers in $H$ in the feasible schedule $\sigma$ (thus the algorithm only rejects some *sections* in $\sigma_t$ to construct

[4]If $\mathcal{A}$ does not consider a section's handler for feasibility (e.g., [5], [12]), then the handler's execution time is regarded as zero.

$\sigma$). Thus, $S_i$ is included in a feasible schedule at time $t$, resulting in zero best-case NBI.

The best-case NBI scenario for DASA, LBESA, and AUA is the same as their worst-case. ■

Thus, HUA's NBI interval $[0, \max_{\forall S_j \in \sigma_t} S_j.X + S_j^h.X]$ lies in between that of DASA/LBESA's $[0]$ and AUA's $[+\infty]$. Note that HUA and AUA bound handler completions; DASA/LBESA do not.

HUA produces optimum total utility for a special case:

*Theorem 4:* Consider a set of threads with step TUFs and no node failures. Suppose there is sufficient processor time for meeting the termination-times of all thread sections and their handlers on all nodes. Now, a system-wide EDF schedule is produced by HUA, yielding optimum total utility.

*Proof:* The theorem is self-evident. For brevity, we skip the proof. ■

## IV. The TPR Protocol

### A. Overview

The TPR protocol is an extension of the Alpha Thread Polling protocol [7]. The protocol is instantiated in a software component called the Thread Integrity Manager (or TIM). Every node which hosts thread sections has a TIM component, which continually runs TPR's three-phase polling operation.

The TPR specifies unique behaviors for nodes hosting the root section of a thread. The TIM on each node is responsible for maintaining the health and coordinating any cleanup required for threads rooted there. Downstream sections, 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 section deems itself an orphan and commences autonomous cleanup. Once this occurs, the thread section is effectively disconnected from the remainder of the thread's call-graph, and control is returned to application code in the context of the section exception handler.

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

### B. Thread Polling

In the first phase, the root node of a given thread regularly broadcasts a ROOT_ANNOUNCE message to all nodes within the system. The ROOT_ANNOUNCE message is sent every $t_p$, or polling interval. Figure 3 illustrates the polling process for a healthy thread.

*Lemma 5:* Under TPR, if a section $S_i$ does not receive a ROOT_ANNOUNCE message within $t_p + D$, then either the root node has failed or the segment has become disconnected. $S_i$ is thus *orphaned.*

*Proof:* Since ROOT_ANNOUNCE message is sent every $t_p$, and $D$ is the worst-case message latency, every healthy section of a healthy thread will receive ROOT_ANNOUNCE within $t_p + D$. ■

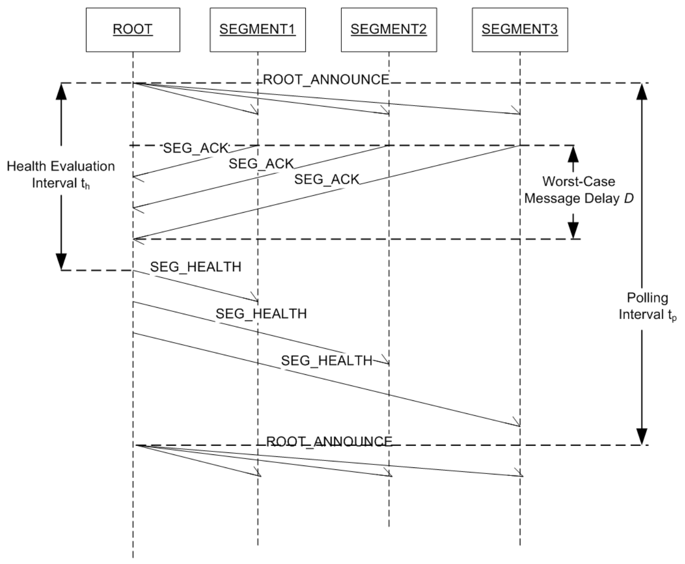

Fig. 3. TPR Operation — Healthy Thread

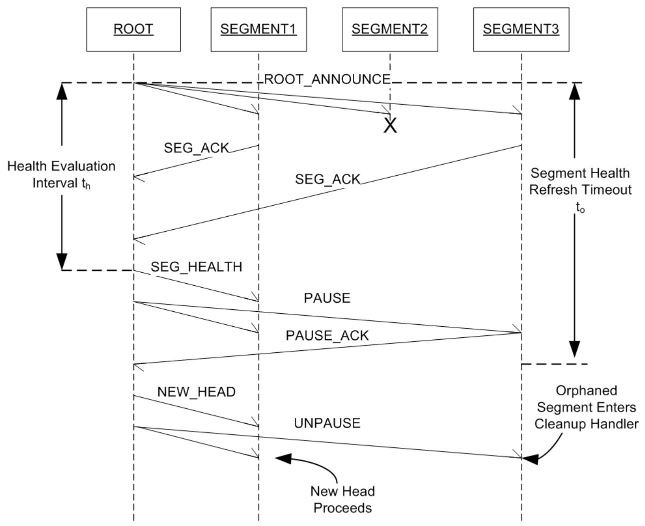

Fig. 4. TPR Operation — Unhealthy Thread Entering Recovery

In the second phase, all nodes that are hosting sections of a thread respond to the ROOT_ANNOUNCE with a section acknowledgment (SEG_ACK) message. The root node will receive a SEG_ACK message from every healthy section within a delay of $2D$ following a ROOT_ANNOUNCE broadcast. This delay, called the *thread health evaluation time* $t_h \geq 2D$ may be tuned as a function of the worst-case message delay to ensure that no acknowledgment messages are missed.

In the last phase, the root node waits for $t_h$ to expire before examining the information it has received from the SEG_ACK messages to determine the thread's status (broken or unbroken).

*Lemma 6:* Under TPR, the root node will detect a broken thread within $t_p + t_h$, where $t_h \geq 2D$.

*Proof:* The worst-case scenario for detecting a broken thread occurs when a node fails immediately after sending a SEG_ACK. Thus, the root node will miss discovering the thread break within $t_h$ of the ROOT_ANNOUNCE broadcast, and must wait for the next thread health evaluation time $t_h$ to elapse to detect the break. The next health evaluation time will start no later than one $t_p$. The lemma follows. ■

If the thread is determined to be unbroken, then the root sends health update (SEG_HEALTH) messages to all sections of the thread, refreshing them. If there is a break in the thread, the root node refreshes only sections of the thread deemed healthy, and enters the recovery state to deal with the break.

*Lemma 7:* Under TPR, every healthy section of a healthy thread will receive a SEG_HEALTH message at a maximum interval of $t_p + t_h + D$.

*Proof:* A root node broadcasts a ROOT_ANNOUNCE message every $t_p$ and determines a thread's status after $t_h$. Following this, it sends a SEG_HEALTH message to all healthy sections of the thread. Since the worst-case message latency is $D$, every healthy section of a healthy thread will receive a SEG_HEALTH message within $t_h + D$ of the receipt of a ROOT_ANNOUNCE message. The lemma follows. ■

## C. Recovery

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

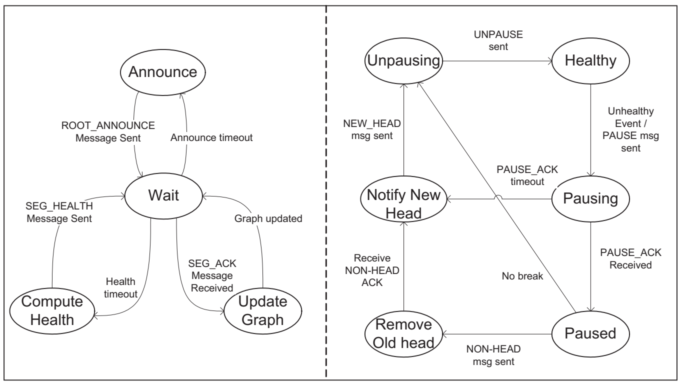

Fig. 5. High-level State Diagram – Root Section

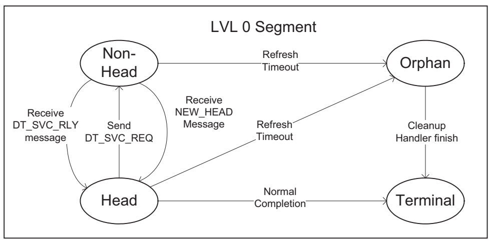

Fig. 6. High-level State Diagram – Section

Recovery from a thread break proceeds through four steps: 1) Pausing the thread and waiting for pause acknowledgment; 2) Determining which section will be the new head; 3) Notifying the new head section that it may continue to execute; and 4) Unpausing the thread.

Figure 4 illustrates the protocol operation for recovering from an unhealthy thread.

Figure 5 illustrates the states experienced by an individual thread from the standpoint of its root section; Figure 6 illustrates the states from the standpoint of a section. 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 section from the root. This section 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.

*Lemma 8:* Once a thread failure is detected, TPR activates a new thread head within $t_p + t_h + 4D$.

*Proof:* By Lemma 6, the root will detect a broken thread within $t_p + t_h$. Subsequently, the thread is paused within $2D$ ($D$ for sending PAUSE and $D$ for receiving PAUSE_ACK), and a new head is activated within another $2D$ ($D$ for sending NEW_HEAD and $D$ for sending UNPAUSE). The lemma follows. ■

From here, the point of execution is to 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 on how to proceed (e.g., resumption of thread execution).

### D. Orphan Cleanup

When a section 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 TIM determines which locally hosted sections, if any, are orphans. The manager then schedules the respective exception handler code to be run for each orphan. Orphan cleanup serves both to remove sections that follow a break in the thread (called *thread trimming*) and to remove the entirety of threads that have lost their root.

*Theorem 9:* If threads are scheduled using HUA, then every unhealthy section $S_i$ will detect that it is an orphan and clean up within $t_p + t_h + D + S_i.X + S_i^h.X$.

*Proof:* Lemma 7 implies that every unhealthy section $S_i$ will detect that it is an orphan within $t_p + t_h + D$. Theorem 1 implies that $S_i$'s handler will complete within $S_i.X + S_i^h.X$, once $S_i$ fails per Definition 1. Definition 1 subsumes the case of $S_i$ receiving a notification regarding the failure of an upstream section, which implies that $S_i$ has become an orphan. The theorem follows. ■

## V. Implementation Experience

We implemented HUA and TPR in the emerging Reference Implementation (RI) of DRTSJ [1]. The RI includes a user-space scheduling framework for pluggable thread scheduling (similar to [11]) and mechanisms for implementing thread integrity protocols (e.g., TIM). The RI infrastructure runs atop Apogee's Real-Time Java Virtual Machine that is compliant with the Real-Time Specification for Java (RTSJ). This RTSJ platform runs atop the Debian Linux OS (kernel version 2.6.16-2-686) on a 800MHz, Pentium-III processor. Our experimental testbed consisted of a network with five such DRTSJ nodes.

Besides HUA, we also implemented the AUA algorithm [6]. This allows a comparison between HUA/TPR and AUA/TPR (described in [6]). Our test application was composed of one master node and four slave nodes. The master node was responsible for issuing commands to the slave nodes and logging events on a single timescale. The slave nodes were required to accept commands from the master node and were responsible for the execution, propagation, and maintenance of threads.

Our metrics of interest included the Total Thread Cleanup Time, the Failure Detection Time, New-Head Notification Time, the Handler Completion Time, and the measured NBI. We measured these during 100 experimental runs of the test application. Each experimental run spawned a single distributable thread, which propagated to five other nodes and then returned back through the same five nodes.

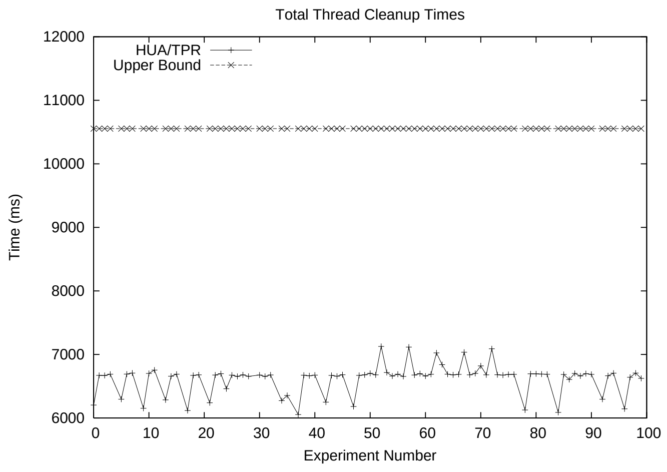

Fig. 7. Total Thread Cleanup Times

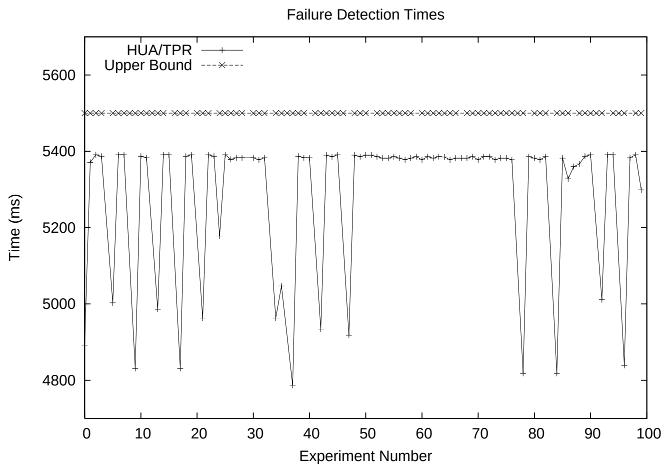

Fig. 8. Failure Detection Times

The Total Thread Cleanup Time is the time between the failure of a thread's node (causing a section failure) and the completion of the handlers of all the orphan sections of the thread. Figure 7 shows the measured cleanup time for HUA/TPR plotted against its cleanup upper bound time for the thread set used in our experiments. We observe that HUA/TPR satisfies its cleanup upper bound, validating Theorem 9.

In order for the Total Thread Cleanup Time to satisfy the HUA/TPR cleanup bound, TPR must detect a failure within a certain amount of time as determined by the protocol parameters (e.g., polling interval $t_p$). Figure 8 shows TPR's Failure Detection times as measured during failures in our test application and the upper bound on failure detection time as calculated using our experimental parameters. As the figure shows, TPR satisfies the upper bound on failure detection.

The variation observed in Figure 8 is actually less than the theoretic variation of $t_p$ (1 second for these experiments) as described in Lemma 6. This variation also occurs in Figure 9 for the same reason.

For a thread to recover from a thread break, a new head must be established and orphans must be notified to clean themselves up. Therefore, the last time that must be bounded in order for HUA/TPR to achieve an upper limit on orphan cleanup is the time it takes for the protocol to determine and notify a thread of its new head. We measure this as the New-Head Notification Time. Figure 9 shows TPR's New-Head Notification Time and the notification time bound that TPR must satisfy in order to meet the Total Thread Cleanup bound. We observe that HUA/TPR satisfies the notification time bound.

Figure 10 shows the thread completion times of experiments 1) with failures and TPR, 2) without failures and without TPR, 3) without failures and without TPR, and 4) with failures and without TPR. By measuring the thread completion times under these scenarios, we measure the overhead of TPR in terms of the increase in thread completion times caused by the protocol operation.

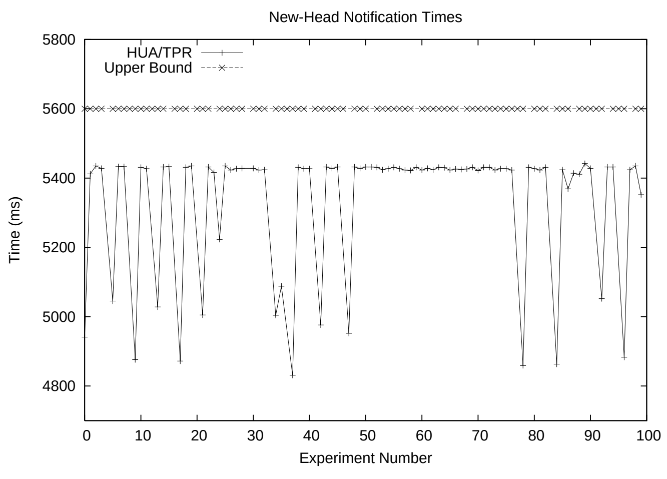

Fig. 9. New-Head Notification Times

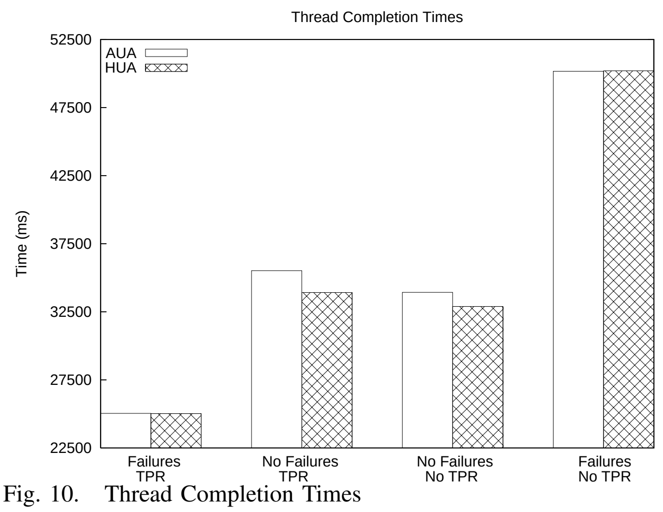

Fig. 10. Thread Completion Times

Thread Completion Time is the difference between the time when a root section of a thread starts and the time when it completes. As orphan cleanup can occur in parallel with the continuation of a repaired thread, Thread Completion Time may ignore orphan cleanup times, making completion times of failed threads shorter than completion times of successful threads. This behavior is evident in Figure 10 as the experiments with failures and with TPR had the shortest completion times.

One of the most interesting aspects of Figure 10 is the contrast between the experiments without failures. This contrast shows the overhead that TPR incurs when there are no failures present. Another interesting aspect of the figure is the large completion times for experiments with failures, but without TPR. The DRTSJ platform that we used for implementing HUA/TPR enforces a simple, tunable failure detection scheme in the absence of a thread integrity protocol. We purposely chose a large failure detection delay to convey the idea that the threads would never complete without any kind of failure detection and are subject to longer than necessary completion times if the detection scheme is naive.

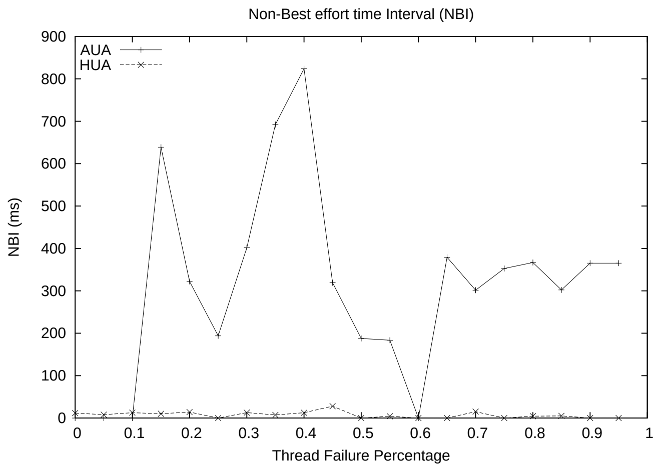

Fig. 11. Non-Best-effort time Interval (NBI)

Figure 11 shows the NBI of AUA and HUA under increasing number of thread failures. We observe that AUA has a higher NBI than HUA, validating Theorems 2 and 3. This difference in NBI is due to AUA's admission control policy [6]: AUA rejects threads arriving during overloads to respect the assurance made to previously admitted threads, irrespective of thread importance. HUA does not have this policy, and is generally free (limited by its NBI) to admit threads arriving during overloads that might have higher PUDs.

The variation of the NBI observed in Figure 11 is due to the way failures were experimentally created. The sets of failed threads were identical for all experiments at the same failure percentage. But, they were not a strict subset of the sets of failed threads for experiments with higher failure percentages.

## VI. Conclusions and Future Work

We present a real-time scheduling algorithm called HUA and a distributable thread integrity protocol called TPR. HUA/TPR's application model includes distributable threads and their exception handlers with TUF time constraints, and an execution paradigm where handlers abort the failed threads after performing recovery actions. We show that HUA/TPR bounds the orphan cleanup and recovery time with bounded loss of the best-effort property. Our implementation experience of HUA/TPR using the emerging RI of Sun's DRTSJ demonstrates the algorithm/protocol's effectiveness.

Directions for future work include considering mobile, ad-hoc networks, relaxing the upper bound on message latency, and relaxing the requirements for reliable communication.

## References

[1] J. Anderson and E. D. Jensen. The distributed real-time specification for java: Status report. In *JTRES*, 2006.

[2] A. Bestavros and S. Nagy. Admission control and overload management for real-time databases. In *Real-Time Database Systems: Issues and Applications*, chapter 12. Kluwer Academic Publishers, 1997.

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

[4] R. Clark, E. D. Jensen, et al. An adaptive, distributed airborne tracking system. In *IEEE WPDRTS*, pages 353–362, April 1999.

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

[6] E. Curley, J. S. Anderson, B. Ravindran, and E. D. Jensen. Recovering from distributable thread failures with assured timeliness in real-time distributed systems. In *IEEE SRDS*, pages 267–276, 2006.

[7] J. Goldberg, I. Greenberg, et al. Adaptive fault-resistant systems (chapter 5: Adpative distributed thread integrity). Technical Report csl-95-02, SRI International, January 1995. `http://www.csl.sri.com/papers/sri-csl-95-02/`.

[8] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quaterly*, 21:177–185, 1974.

[9] E. D. Jensen et al. A time-driven scheduling model for real-time systems. In *IEEE RTSS*, pages 112–122, Dec. 1985.

[10] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE TPDS*, 8(12):1268–1274, Dec. 1997.

[11] 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.

[12] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, CMU, 1986. CMU-CS-86-134.

[13] D. P. Maynard, S. E. Shipman, et al. An example real-time command, control, and battle management application for alpha. Technical Report Archons Project Technical Report 88121, CMU CS Dept., December 1988.

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

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

[16] OMG. Real-time corba 2.0: Dynamic scheduling specification. Technical report, Object Management Group, September 2001.

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

[18] H. Streich. Taskpair-scheduling: An approach for dynamic real-time systems. *Mini and Microcomputers*, 17(2):77–83, 1995.

[19] The Open Group. *MK7.3a Release Notes*. The Open Group Research Institute, Cambridge, Massachusetts, October 1998.