Integrity Protocols for Recovering from Distributable Real-Time Thread Failures with Assured Timeliness in Dynamic Systems

Binoy Ravindran*, Edward Curley*, Jonathan Anderson*, and E. Douglas Jensen‡

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

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

Abstract

We consider the problem of recovering from failures of distributable threads with assured timeliness in dynamic systems with overloads, and node and (permanent/transient) network failures. When a distributable thread encounters a failure that prevents its timely execution, the thread must be terminated. Thread termination involves detecting and aborting thread orphans, and delivering exceptions to the farthest, contiguous surviving thread segment for possible execution resumption. Thread termination operations must optimize system-wide timeliness. We present a scheduling algorithm called HUA and two thread integrity protocols called D-TPR and W-TPR. We show that they bound the orphan cleanup and recovery time with bounded loss of the best-effort property—i.e., high importance threads are always favored over low importance ones (for feasible completion), irrespective of thread urgency. Our implementation experience using the emerging Reference Implementation of Sun’s Distributed Real-Time Specification for Java demonstrates the algorithm/protocols’ effectiveness.

1. Introduction

In distributed systems, action and information timeliness is often end-to-end—e.g., a causally dependent, multi-node, sensor to shooter sequential flow of execution in network-centric warfare systems [3]. Designers and users of distributed systems often need to dependably reason about (specify, manage, predict) end-to-end timeliness. Many emerging such systems are being envisioned to be built using ad hoc network systems—e.g., those without a fixed network infrastructure and have dynamic node membership and network topology changes, including mobile, ad hoc wireless networks [2]. Reasoning about timeliness, especially end-to-end, is a very difficult and unsolved problem in such dynamic uncertain systems.

Maintaining end-to-end properties (e.g., timeliness, connectivity) of a control or information flow requires a model of the flow’s locus in space and time that can be reasoned about. Such a model facilitates reasoning about, and resolving the contention for resources that occur along the flow’s locus. The distributable thread abstraction which first appeared in the Alpha OS [13] and later in MK7.3 [17], OMG’s Real-Time CORBA 1.2 [14], and Sun’s emerging Distributed Real-Time Specification for Java (DRTSJ) [1] directly provides such a model as their first-class programming and scheduling abstraction. A distributable thread is a single thread of execution with a globally unique identity that extends and retracts through local and remote objects. We focus on distributable threads as our end-to-end control flow/scheduling abstraction, and hereafter, refer to them 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. 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 threads [14].

We consider the problem of developing assurances on thread timing behavior in dynamic systems, in the presence of application/network-induced uncertainties. The uncertainties include transient and sustained resource overloads (due to context-dependent thread execution times), arbitrary thread arrivals, node failures, and transient and permanent link failures (causing varying packet drop rate behaviors). 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—e.g., if all time constraints are deadlines, then EDF [8] can meet all deadlines.)

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 its completion time. We specify a deadline as a binary-valued, downward “step” shaped TUF; Figure 2 shows examples. A thread’s TUF decouples its importance and urgency—urgency is measured on the X-axis, and importance is denoted (by utility) on the Y-axis.

When thread time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued 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 (e.g., [5, 11]).

UA algorithms that maximize total utility under downward step TUFs (e.g., [5, 11]) default to EDF during underloads, since EDF satisfies all deadlines during underloads. Consequently, they obtain the optimum total utility during underloads. During overloads, they inherently favor more important threads over less important ones (since more utility can be attained from the former), irrespective of thread urgency, and thus exhibit adaptive behavior and graceful timeliness degradation. This behavior of UA algorithms is called “best-effort” [11] 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. Thus, high importance threads that arrive at any time always have a very high likelihood for feasible completion (irrespective of their urgency).

Our Contributions. When nodes transited by threads fail, it can divide those threads 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 fail and cause 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, thread execution can potentially be resumed 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.

Scheduling of the orphan-clean up 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 model them as traditional (single-node) threads, since these are similar in nature, with similar scheduling parameters such as execution time and time constraints. Further, the classical admission control strategy [6, 12, 16] can be used: When a thread T arrives on a node, if a feasible node schedule can be constructed such that it includes all the previously admitted threads and their handlers, besides T and its handler, then admit T and its handler; otherwise, reject. But this will cause the very fundamental problem that is solved by UA schedulers through their 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.

In this paper, we consider the problem of recovering from thread failures with assured timeliness and best-effort property. 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 and a network with permanent/transient failures and unreliable transport mechanisms. For such a model, we consider the scheduling objective of maximizing the total accrued utility.

We present a UA scheduling algorithm called Handler-assured Utility Accrual scheduling algorithm (or HUA) for thread scheduling, and two protocols called Decentralized Thread Polling with bounded Recovery (or D-TPR) and Wireless Thread Polling with bounded Recovery (or W-TPR) for ensuring thread integrity. D-TPR targets networks with generally permanent network failures, and W-TPR targets mobile, ad hoc wireless networks with generally transient network failures. We show that HUA and D-TPR/W-TPR ensure that handlers of threads that encounter failures during their execution will complete within a bounded time, yielding bounded thread cleanup time. Yet, the algorithm/protocols retain 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 using DRTSJ’s emerging Reference Implementation (RI) demonstrates the algorithm/protocols’ effectiveness.

Similar to UA algorithms, thread integrity protocols have been developed in the past—e.g., Thread Polling with bounded Recovery [6], Alpha’s Thread Polling [13], Node Alive protocol [7]. However, none of these efforts provide time-bounded thread cleanup in the presence of node and (permanent/transient) network failures and unreliable transport mechanisms. Further, [6] suffers from unbounded loss of the best-effort property due to its admission control strategy (we show this in Section 3.3). In contrast, HUA
and D-TPR/W-TPR provide bounded thread cleanup with bounded loss of the best-effort property in the presence of (permanent/transient) network failures and unreliable transport mechanisms — the first such algorithm/protocols. Thus, the paper’s contribution is the HUA and D-TPR/W-TPR.

The rest of the paper is organized as follows: In Section 2, we state our models and objectives. Section 3 presents HUA, Section 4 presents D-TPR, and Section 5 presents W-TPR. In Section 6, we discuss our implementation experience. We conclude the paper in Section 7.

2. Models and Objectives

Threads. 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.

A section’s execution time estimate is known when the thread arrives at the section’s node. This execution 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).

A thread’s total number of sections is unknown a-priori, as the 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 \( T = \{ T_1, T_2, T_3, \ldots \} \).

Timeliness Model. Each thread \( T_i \)’s time constraint is specified using a TUF, denoted \( U_i(t) \). A classical deadline is unit-valued — i.e., \( U_i(t) = \{0,1\} \), since importance is not considered. Downward step TUFs (Figure 2) generalize classical deadlines where \( U_i(t) = \{0,\{n\}\} \). We focus on non-increasing (unimodal) TUFs, as they encompass the majority of time constraints of interest to us (e.g., [4]).

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 \( U_i(t) > 0, \forall t \in [I_i, X_i] \) and \( U_i(t) = 0, \forall t \notin [I_i, X_i], \forall i \).

Abort Model. Each section of a thread has an associated exception handler. We consider a termination model for all thread 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 hosting thread’s sections. When a handler executes, it will abort the associated section after performing recovery actions that are necessary to avoid inconsistencies — e.g., rolling back/forward section’s held logical and physical resources to safe states.

We consider a similar abort model for node and network failures. When a thread encounters a node/network failure causing orphans, an integrity protocol (e.g., D-TPR) delivers failure-notification messages to all the orphan nodes. Those nodes then respond by releasing handlers which abort the orphans after executing recovery 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.

System and Failure Models. We consider a system model, where a set of processing components, generically referred to as nodes, \( N_i \in N, i \in [1, m] \), are interconnected via a network. We consider a multipath network model (e.g., WAN, MANET), with nodes interconnected through routers. Node clocks are synchronized — e.g., using [15].

Network is assumed to be unreliable. Nodes may fail arbitrarily by crashing (i.e., fail-stop). Network links may fail transiently or permanently, causing network partitions, again, arbitrarily. Only an unreliable message transport protocol like UDP is assumed. We describe thread integrity protocol-specific network assumptions in Sections 4 and 5.

Each node executes thread sections. The order of executing sections on a node is determined by the node scheduler. We consider Real-Time CORBA 1.2’s [14] Case 2 approach for thread scheduling. According to this approach, node schedulers use the propagated thread scheduling parameters and independently schedule thread sections to optimize the system-wide timeliness optimality criterion. Though this results in approximate, global, system-wide timeliness, Real-Time CORBA supports the approach due to its simplicity and capability for coherent end-to-end scheduling.

Scheduling Objectives. Our primary objective is to maximize the total thread accrued utility 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 at which all orphans of the thread complete. The algorithm must also exhibit the best-effort property of UA algorithms (Section 1) to the extent possible.

3. The HUA Algorithm

3.1. 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, node (section) schedules must be constructed solely exploiting the current system knowledge. Since the 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) [5]. On a node, a section’s PUD measures the utility that can be accrued per unit time by immediately executing it on the node.

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

**Definition 1 (Section Failure).** Consider a section $S_i$ of a 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 $S_i$, which designates $S_i$ as an “orphan-head.”

**Definition 2 (Released Handler).** A handler is 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. The section’s PUD can now be measured as the minimum of these two PUDs, as that is the worst-case.

Thus, on each node, HUA examines thread sections for potential inclusion in a feasible node schedule 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 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 its thread $T_i$; if so, $S_i$ is reconsidered for scheduling 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 integrity protocol (e.g., D-TPR).

This process ensures that the sections that are included in a node’s schedule at any time have feasible handlers. Further, all their upstream sections also have feasible handlers on their respective nodes. Thus, when any such section fails (per Definition 1), its handler and the handlers of all its upstream sections will 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. Thus, $S_i$ 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 the total number of sections of a thread is unknown. If a thread’s total number of sections is known a-priori, then better schemes (e.g., [10]) that intelligently distribute the thread’s total slack 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 $S_j$ is released on a node after the failure of another section $S_i$ at the node (per Definition 1) and before the completion of $S_j$’s handler on the node. As per the best-effort philosophy, $S_j$ 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 $S_j$ for execution until $S_j$’s handler completes, consequently degrading the algorithm’s best-effort property. In Section 3.3, we quantify this loss.

### 3.2. Algorithm Overview

HUA’s scheduling events at a node include the arrival of a thread at the node, release of a handler 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):

---

\[ \text{PUD} \]
\[ S_i \] is the current set of unscheduled sections including a newly arrived section (if any). \( S_i \in S_i \) is a section. \( S_i^h \) denotes \( S_i \)'s handler. \( T_j \) denotes the thread to which a section \( S_i \) and \( S_i^h \) belong. \( S_i.X \) is \( S_i \)'s termination time, which is the same as that of \( T_j \)'s termination time. \( S_i.ExT \) is \( S_i \)'s estimated remaining execution time. \( U_i(t) \) denotes \( S_i \)'s TUF, which is the same as that of \( T_j \)'s TUF. \( U_i(t) \) denotes \( S_i^h \)'s TUF.

- \( \sigma \) is the schedule (ordered list) constructed at the previous scheduling event. \( \sigma \) is the new schedule.
- \( H \) 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.
- The 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.
- The function alertProtocol() declares \( S_i \) as failed (e.g., with D-TPR, this is done by \( S_i \)'s node not sending POLL messages to \( S_i \)'s predecessor and successor section nodes).
- The function IsHead() returns true if \( S \) is a head; false otherwise.
- The function headOf() returns the first section in \( \sigma \).
- The function sortByPUD() 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 \( ExT \) will appear before any others with the same PUD.
- The function Insert() inserts section \( S \) in the ordered list \( \sigma \) at the position indicated by index \( I \); if entries in \( \sigma \) exist with the index \( I \), \( S \) is inserted before them. After insertion, \( S \)'s index in \( \sigma \) is \( I \). Remove() removes section \( S \) from ordered list \( \sigma \) at the position indicated by index \( I \).
- The function feasible() 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 \( ExT \)s of all sections that occur before \( S \) in \( \sigma \) and \( S.ExT \).

Algorithm 1 describes HUA at a high level of abstraction. When invoked at time \( t_{\text{curr}} \), 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).

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 \( S_i \) and \( S_i^h \) are inserted into \( \sigma \) at positions that correspond to \( S_i.X \) and \( S_i.X + S_i^h.X \), respectively.

If after inserting \( S_i \) and \( S_i^h \) into \( \sigma \), \( \sigma \) becomes infeasible, \( S_i \) and \( S_i^h \) are removed (lines 13–14). If a section \( S_i \) that is removed is not a head and belonged to the schedule constructed at the previous scheduling event, the integrity 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).

### 3.3. Algorithm Properties

**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 \( S_i \) is in HUA’s current schedule. This implies that both \( S_i \) and \( S_i^h \) were feasible, and \( S_i^h \) was scheduled to complete by \( S_i.X + S_i^h.X \).

If \( S_i \) receives a notification on the failure of an upstream section \( S_j \) at a time \( t \), then all sections from \( S_j \) to \( S_i \) and their handlers are feasible on their nodes, as otherwise the thread execution would not have progressed to \( S_i \). Thus, \( S_i^h \) is scheduled to complete by \( S_i.X + S_i^h.X \). A similar argument holds for the case of \( S_i \) receiving a notification on the failure of a downstream section. Theorem follows.

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 \( A \). 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 \( A \)'s schedule at
time $t$ are not feasible at $t$, but $S_i$ and its handler are feasible just by themselves; (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 $A$’s schedule at time $t$. Now, $A$’s NBI, $NBIA$, is defined as the duration of time that $S_i$ will have to wait after $t$, before it is included in $A$’s feasible schedule. Thus, $S_i$ is assumed to be feasible together with its handler at $t + NBIA$.

We now describe the NBI of HUA and other UA algorithms including DASA [5], LBESA [11], and AUA [6] (under zero overhead):

**Theorem 2.** HUA’s worst-case NBI is $t + \max_{\sigma \in S_i \in \sigma} (X + S_{ij}^b \times X)$, where $\sigma$ is 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 $\sigma = H \neq \emptyset$. Since $S_i$ has the highest PUD and is feasible, $S_i$ 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 $\sigma$ complete. Since each handler in $\sigma$ is scheduled to complete by $\max_{\sigma \in S_i \in \sigma} (X + S_{ij}^b \times X)$, the earliest time that $S_i$ becomes feasible is $t + \max_{\sigma \in S_i \in \sigma} (X + S_{ij}^b \times X)$.

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 $+\infty$. $\Box$

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

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

$\Box$

4. The D-TPR Protocol

D-TPR targets systems with node and network failures that are generally permanent. The protocol is instantiated in a software component called the Thread Integrity Manager (or TIM). Every node hosting thread sections has a TIM, which continually runs D-TPR’s polling operation.

The TIM’s operations 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. For simplicity in analysis, we also assume perfectly synchronized clocks.

![D-TPR Healthy Operation](image)

Figure 3 shows a sequence diagram for the operation of D-TPR for a healthy thread.

4.1. Polling

At every polling interval $t_p$, the TIM on each node identifies the sections that are locally hosted. The TIM then sends a POLL message to each of its predecessor and successor nodes. Note that each node can host sections of several threads so a single node may have several predecessor and successor nodes.

Each POLL message (see Table 1) is a list of entries, where each entry contains a type, the local section ID the entry corresponds to, and a remote section ID. If the entry type is SUCCESSOR, the remote section ID will correspond to the successor section of the local section in the entry. Similarly, the remote section ID of PREDECESSOR corresponds to the predecessor section of the local segment in the entry. In this way, the node receiving the POLL message is able to discern (downstream or upstream) the message’s origin and thus from which direction the section has been deemed healthy. This distinction becomes important for break detection and is discussed further.

4.2. Break Detection

When an invocation is made, D-TPR creates two timers which are set to a delay $D$. $D$ is assumed to be the delay incurred in successfully transporting a message from one node to another in the network with very high probability, and is empirically determined (similar to our measurements in Section 6).3 One timer is established for the downstream section and the other is established for the upstream section. The TIM on the node making the invocation (upstream side) creates a downstream-invocation timer that will cause a timeout when polling messages have not been received from downstream frequently enough. The TIM on the node hosting the remote object to which the invocation is being made (downstream side) creates an upstream-invocation timer that will cause a timeout when polling messages are not received from upstream frequently enough.

3Thus, the lack of the receipt of a message at a destination node $N_j$ within $D$ of sending the message from a node $N_i$ is considered a network failure—e.g., $N_j$ is unreachable from $N_i$; a network partition between $N_i$ and $N_j$—with high probability.
When a POLL message is received from upstream, the upstream-invocation timer is reset to \( D \) and resumes counting down. The same is true of the downstream-invocation timer when a POLL message is received from downstream. A “thread break” is declared when either the upstream or downstream-invocation time reaches zero. Recovery is different depending on which timer experiences the timeout.

**Lemma 4.** Consider a section \( S_i \) and its successor section \( S_j \). Under D-TPR, if \( S_i \)'s node fails, or \( S_i \) becomes unreachable from \( S_j \) (but not necessarily vice versa), then \( S_j \) will detect a thread break between \( S_i \) and \( S_j \) within \( t_p + D \).

**Proof.** TPR’s worst-case scenario for detecting this thread break occurs when \( S_j \)'s node crashes immediately after \( S_j \) sends the POLL message to \( S_i \) (and the network successfully delivers that POLL to \( S_i \)), or when \( S_i \) becomes unreachable from \( S_j \) immediately after \( S_i \) receives \( S_j \)'s POLL message. Consequently, \( S_i \) will miss discovering the thread break when it receives the POLL, and must wait for the lack of the next POLL from \( S_j \) to detect the break. The next POLL will be sent no later than one \( t_p \), the lack of the receipt of which will be detected by \( S_i \) no later than one \( D \). The lemma follows.

**Lemma 5.** Consider a section \( S_j \) and its predecessor \( S_i \). Under D-TPR, if \( S_i \)'s node fails, or \( S_i \) becomes unreachable from \( S_j \) (but not necessarily vice versa), then \( S_j \) will detect a thread break between \( S_i \) and \( S_j \) within \( t_p + D \). \( S_j \) and its downstream sections are now said to be orphaned.

**Proof.** The proof is similar to that of Lemma 4.

### 4.3. Recovery

D-TPR’s recovery operations are administrative functions, and carries on below the level of application scheduling. While recovery proceeds, D-TPR activities continue concurrently. This allows the protocol to recognize and deal with multiple simultaneous breaks and cleanup operations.

If the upstream-invocation timer expires, the protocol assumes that the upstream section is unreachable and declares the local section associated with the timer to be an orphan. D-TPR then attempts to accomplish two things: first, force the upstream section to become the thread’s new head; and second, force the downstream section to become an orphan.

To force the upstream section to become the new head, the protocol sends a NEW HEAD message upstream and stops upstream POLL messages, which refresh the upstream section. If the upstream node receives the NEW HEAD message, the upstream section will immediately begin behaving like a new head. If the upstream node does not receive the message, the upstream section’s downstream-invocation timer will expire (due to the stopped POLL messages) forcing the section to become the new head.

In order to force the downstream section to become an orphan, the protocol sends an ORPHAN PROP message downstream and modifies its downstream POLL messages to include an orphan status. The downstream node will either receive the ORPHAN PROP message and become an orphan, or the downstream section’s timer will expire forcing it to become an orphan. When a section becomes an orphan, it propagates the ORPHAN PROP message in order to identify all orphans. Figure 4 shows this scenario.

### Table 1. D-TPR Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Contents</th>
<th>From/To</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLL</td>
<td>List of local section ID and remote ID pairs. Remote section IDs are either predecessor or successor sections to local section</td>
<td>travel back and forth between predecessor and successor nodes</td>
</tr>
<tr>
<td>NEW HEAD</td>
<td>timed out section and predecessor section</td>
<td>node with upstream timeout to predecessor node</td>
</tr>
<tr>
<td>ENDORPHAN</td>
<td>timed out section and successor section</td>
<td>node with downstream timeout to successor node</td>
</tr>
<tr>
<td>ORPHAN PROP</td>
<td>orphaned section and successor section</td>
<td>node with orphan section to successor node</td>
</tr>
</tbody>
</table>

![Figure 4. D-TPR Unhealthy Upstream](image)

When a section’s downstream-invocation timer expires, the protocol assumes that the downstream sections are unreachable and declares itself the new head of the thread. The new head then sends an ENDORPHAN downstream and ceases downstream refresh polling. In this way, the downstream section will either receive the ENDORPHAN notification and become an orphan or its upstream timer will expire, making the section an orphan. Figure 5 shows this scenario.

**Lemma 6.** Under D-TPR, if a thread break occurs between \( S_i \) and its successor \( S_j \), then \( S_i \) will become the new head within \( t_p + 2D \). Since the new head of a thread is always

![Figure 5. D-TPR Unhealthy Downstream](image)
directly upstream from a break, D-TPR therefore activates a new head within \(t_p + 2D\).

**Proof.** A thread break between \(S_i\) and \(S_j\) can occur in primarily two ways: (I) \(S_i\)’s node fails, or \(S_i\) becomes unreachable from \(S_j\); and (II) \(S_j\)’s node fails, or \(S_i\) becomes unreachable from \(S_j\). Lemma 4 identifies Case (I). Thus, \(S_i\) will detect the thread break within \(t_p + D\), and immediately after, \(S_i\) will declare itself as the new head, within \(t_p + D\). Case (II) is identified in Lemma 5. Thus, \(S_j\) will detect the break within \(t_p + D\) and will send a NEWHEAD message to \(S_i\). Upon receipt of this message, \(S_i\) will declare itself as the new head, within a total of \(t_p + 2D\) (after break detection), which is the worst-case. Lemma follows.

**Lemma 7.** Under D-TPR, if a thread break occurs between \(S_i\) and its successor \(S_j\), then \(S_j\) will identify itself as an orphan within \(t_p + 2D\).

**Proof.** Proof is similar to that of Lemma 6. A thread break between \(S_i\) and \(S_j\) can occur in primarily two ways: (I) \(S_i\)’s node fails, or \(S_j\) becomes unreachable from \(S_i\); and (II) \(S_j\)’s node fails, or \(S_i\) becomes unreachable from \(S_j\). Case (I) is identified in Lemma 5. Thus, \(S_j\) will detect the break within \(t_p + D\) and will immediately declare itself as an orphan, within \(t_p + D\). Case (II) is identified in Lemma 4. Thus, \(S_i\) will detect the thread break within \(t_p + D\), declare itself as the new head, and send an ENDORPHAN message to \(S_j\). Upon receipt of this message, \(S_j\) will declare itself as an orphan, within a total of \(t_p + 2D\) (after break detection), which is the worst-case. Lemma follows.

### 4.4. Cleanup

A section that has been identified as an orphan will release the section’s exception handler for aborting the section (i.e., the orphan) only if it has been designated an “orphan-head.” This can happen in one of three ways: (1) The current head of the thread becomes an orphan; (2) A non-head orphan is returned to by an orphan-head and becomes a new orphan-head; and (3) An orphan’s downstream-invocation timer expires forcing it to become a new orphan-head.

**Theorem 8.** Under D-TPR/HUA, if a thread break occurs between a section \(S_i\) and its successor \(S_j\), then all orphans from \(S_j\) till the thread’s current head \(S_{j+k}\), for some \(k \geq 1\), will be aborted in the LIFO-order—i.e., from \(S_j\) to \(S_i\)—and will complete by \(t_p + (2 + k)D + \sigma^h_{a=0}(S_{j+a}, X + S^h_{j+a}, X)\), unless a section \(S_{j+a}\) becomes unreachable from \(S_{j+a+1}\), \(0 \leq a \leq k - 1\). Let \(P_i\) denote the thread’s execution sequence be: \(\left\{ \cdots S_i, S_{j+1}, \ldots, S_{j+k} \right\}\). From Lemma 7, \(S_i\) will identify itself as an orphan within \(t_p + 2D\). Following this, the ORPHANPROP message will be propagated from \(S_i\) to \(S_{j+k}\) within \(kD\). Thus, \(S_{j+k}\) will become the first orphan-head and thus the first orphan to be aborted, followed by \(S_{j+k-1}\), \(S_{j+k-2}\), until \(S_j\), following the LIFO-order, since \(S_{j+k-a}\) is always returned to by \(S_{j+k-(a-1)}\), \(0 \leq a \leq k\) by the thread’s execution sequence.

By Theorem 1, a section \(S_a\)’s handler will complete within \(S_a.X + S^h_{a}.X\), once it is an orphan-designate. Thus, all sections from \(S_j\) to \(S_{j+k}\) will complete within \(t_p + 2D + kD + \sigma^h_{a=0}(S_{j+a}, X + S^h_{j+a}, X)\), Theorem follows.

If a section \(S_{j+a}\) becomes unreachable from \(S_{j+a+1}\) \((0 \leq a \leq k - 1)\), then \(S_{j+a}\)’s downstream invocation timer will expire before that of \(S_{j+a+1}\), designating \(S_{j+a}\) as an orphan-head before \(S_{j+a+1}\) — the theorem’s exception.

**Theorem 9.** Under D-TPR/HUA, if a thread breaks, then the thread’s orphans will complete within a bounded time.

**Proof.** This theorem follows from Theorem 8, except for the case when a section \(S_{j+a}\) becomes unreachable from \(S_{j+a+1}\) \((0 \leq a \leq k - 1)\) after a break occurs between \(S_i\) and its successor \(S_j\). If \(S_{j+a}\) becomes unreachable from its successor \(S_{j+a+1}\), then \(S_{j+a}\)’s downstream invocation timer will expire within \(t_p + D\) (similar to Lemma 4, where \(S_i \equiv S_{j+a}\) and \(S_j \equiv S_{j+a+1}\), designating \(S_{j+a}\) as orphan-head. By Theorem 1, now \(S_{j+a}\) will cleanup within \(t_p + D + S_{j+a}.X + S^h_{j+a}.X\). Theorem follows.

## 5. The W-TPR Protocol

W-TPR is designed for mobile, ad hoc wireless networks, where communication is assumed to be unreliable and prone to transient failures (D-TPR considers communication failures to be permanent). The protocol exploits the fact that a thread is only adversely affected by a thread break if the head attempts to move across that break. In contrast, D-TPR detects a break and assumes that the break will be permanent; so it preempts the possibility of the head crossing the break by eliminating sections beyond the break point. W-TPR assumes that the breaks are not permanent.

W-TPR differs from D-TPR primarily in the way thread-breaks are determined. In D-TPR, breaks are recognized when communication between two consecutive nodes of a thread fails for longer than the message delay \(D\), with very high probability. In W-TPR, breaks are never actually recognized. Instead, the protocol recognizes when communication errors affect either an invocation or a return (head movement) and provides maintenance accordingly.

![Figure 6. W-TPR Section State Diagram](image)

Figure 6 shows the states and state transitions that a section undergoes in W-TPR. Note that no breaks are ever declared and that a section becomes an orphan only if it receives the ORPHAN message from an upstream section. Sections assume they are healthy until notified otherwise.

**Downstream Head Movement.** During an invocation, a thread section \(S_i\) makes a call on a remote object, which causes a second section, \(S_{i+1}\) to be created on the remote
node. In order for the invocation to be successful, \( S_{i+1} \) must be created and \( S_i \) must be made aware of \( S_{i+1} \).

When an invocation is made, an invocation request is sent downstream and the local section, \( S_i \), begins waiting for invocation verification. The invocation is verified when the local section receives an INV-ACK from the downstream node or a POLL from the downstream node containing the section ID of the remote section (see further). When the invocation is verified, the local section is stopped until the remote section performs a return (head moves upstream).

Figure 7 shows an example of a successful invocation.

![Figure 7. W-TPR Healthy Operation](image)

When the invocation is received by the downstream node, the downstream node attempts to finalize the invocation and sends an INV-ACK message to the upstream section (to quickly notify it of the successful receipt of the invocation message). Following this, the downstream node begins sending periodic POLL messages to the upstream section, at every polling interval \( t_p \). When a healthy section receives a POLL message from an orphan, the healthy section returns an ORPHAN message to the orphan. If the orphan is not the orphan-head, similar to D-TPR, the ORPHAN message is propagated upstream.

Table 2 describes W-TPR messages.

The protocol resends the invocation request until either the invocation is verified, or the protocol deems that communication with the downstream node is not possible by waiting for an application-specified value \( t_n \) to expire and no INV-ACK or a POLL message is received from the downstream node during \( t_n \). If communication with the downstream node is not possible, then the local section maintains head status and the application is notified that the invocation has failed. The TIM also sends an ORPHAN message downstream, in the event that a partial invocation was accomplished—i.e., the downstream node receives the invocation, and the upstream node becomes unreachable from the downstream node. Thus the downstream node's INV-ACK/POLL messages are not received upstream, while thread execution progresses on the downstream node and further downstream.

Figure 8 shows an unhealthy attempt at an invocation caused by an upstream failure on the left and a downstream failure on the right.

![Figure 8. W-TPR Unhealthy Invocation](image)

Lemma 10. Under W-TPR, the location of a thread's head is ambiguous for at most \( t_n \).

Proof. Directly follows discussion.

Upstream Head Movement. When the head is moving from the local node to an upstream node, the local node begins waiting for return verification from the upstream node. When the return message is received by the upstream node, the upstream node sends a return verification message RETURN-ACK downstream to the local node. If the verification is not received within \( t_n \), then the return times-out (see Figure 9), and the protocol forces the return message to be resent. This process is repeated until the handshake successfully completes or the section on the local node violates its timing constraint. Even in the presence of upstream communication errors, the downstream section never becomes an orphan. Since the section has already finished executing and has a healthy return value, it would be fruitless to abort this section before delivering its return value.

Lemma 11. Under W-TPR, a thread's head is never disconnected from the rest of the thread and no new head activation is required.

Proof. Follows directly from the previous discussion. By Lemma 10, after \( t_n \), the head moves downstream after a fully successful invocation. Any fully successful invocation can execute a return. If the upstream node becomes unreachable when the downstream node executes a return, the downstream section has completed its execution (hence it is returning) and is therefore not an orphan.

Cleanup. A section becomes an orphan when it receives the ORPHAN message in response to one of its POLL messages. When the ORPHAN message is received, the section propagates that message downstream and waits for a return from its downstream section to be designated an orphan-head before starting cleanup, as in D-TPR. Cleanup begins when the furthest orphaned section is notified that it is an orphan.

Theorem 12. Under W-TPR, if a section \( S_i \) makes an unsuccessful invocation to its (potential) successor section \( S_j \) (i.e., \( S_j \) will be \( S_i \)'s successor had if the invocation was successful), then all orphans that can potentially be created from \( S_j \) till the thread’s furthest orphaned section \( S_{j+k} \), \( k \geq 1 \), will be aborted in the LIFO-order and will complete within a bounded time under HUA, as long as no further failures occur between \( S_j \) and \( S_{j+k} \).
Table 2. W-TPR Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Contents</th>
<th>From/To</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLL</td>
<td>section ID pair of $S_i$ and $S_{i-1}$</td>
<td>downstream node to upstream node</td>
</tr>
<tr>
<td>INV-ACK</td>
<td>section ID of section attempting invocation</td>
<td>downstream node to upstream node after receiving an invocation</td>
</tr>
<tr>
<td>RETURN-ACK</td>
<td>section ID of section attempting return</td>
<td>upstream node to downstream node after return success</td>
</tr>
<tr>
<td>ORPHAN</td>
<td>section ID pair of $S_i$ (healthy) and $S_{i+1}$ (orphan)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9. W-TPR Unhealthy Return

Proof. By Lemma 10, after $t_n$, $S_i$ retains the head status since the invocation was unsuccessful, and an ORPHAN message is propagated to all downstream sections till $S_{j+k}$. The rest of the proof follows that of Theorem 9.

Note that Theorem 12 holds only if no further failures occur between $S_j$ and $S_{j+k}$. If such a failure were to occur, then the ORPHAN message may not be propagated or an orphan-head may not be able to return to a non-head orphan. D-TPR can detect such failures due to its continuous pairwise polling operation, whereas W-TPR is unable to do so precisely due its “on-demand” polling approach.

Theorem 13. Under W-TPR/HUA, if orphans are created for a thread as in Theorem 12, then all the orphans will complete within a bounded time, as long as no further failures occur between $S_j$ and $S_{j+k}$.

Proof. Follows from Theorem 12.

6. Implementation Experience

We implemented HUA, D-TPR, and W-TPR in DRTSJ’s RI [1]. The RI includes a threads API, user-space scheduling framework for pluggable thread scheduling, and mechanisms for implementing thread integrity protocols (e.g., TIM). The RI infrastructure runs atop Apogee’s Real-Time Specification for Java (RTSJ)-compliant Aphelion Java Virtual Machine. The RTSJ platform runs atop the Debian Linux OS (kernel version 2.6.16-2-686) on a 800MHz Pentium-III machine. Our experimental testbed consisted of a network with five such DRTSJ nodes.

Our metrics of interest included Total Thread Cleanup Time and protocol overhead as measured by Thread Completion Time. We measured these during 100 experimental runs of our test application (application details are omitted for brevity). Each experimental run spawned a single distributable thread, which propagated to five other nodes and then returned back through the same five nodes.

The Total Thread Cleanup Time is the time between the failure of a thread’s node or communication link and the completion of the handlers of all the orphan sections of the thread. Figures 10 and 11 show the measured cleanup times for HUA/D-TPR and HUA/W-TPR, respectively. The cleanup times are plotted against the protocols’ cleanup upper bound times for the thread set used in our experiments. From the figures, we observe that both HUA/D-TPR and HUA/W-TPR satisfy their cleanup upper bound, thereby validating Theorems 9 and 13.

Figure 10. D-TPR Thread Cleanup Times

Thread Completion Time is the difference between when a root section starts execution and when it completes. Thread cleanup time is not taken into consideration here. Figures 12 and 13 show the thread completion times of experiments 1) with failures and D-TPR/W-TPR, 2) without failures but...
with D-TPR/W-TPR, 3) without failures and without D-TPR/W-TPR, and 4) with failures but without D-TPR/W-TPR. By measuring the thread completion times under these scenarios, we measure the overhead each protocol incurs in terms of the increase in thread completion times.

Figure 12 shows the completion times for experiments with and without D-TPR. We observe that the completion times of successful threads without D-TPR is smaller than that with D-TPR. This is to be expected as D-TPR incurs a non-zero overhead. However, we also observe that the completion times of failed threads with D-TPR are shorter than even the completion times of successful threads without D-TPR. This is because, orphan cleanup can occur in parallel with the continuation of a repaired thread, allowing the repaired thread to finish without waiting for all orphans to run to completion. A successful thread, on the other hand, must wait for all sections to finish before it can complete, increasing its completion time. Figure 12 also shows that failed threads with D-TPR complete much more quickly than failed threads with no D-TPR support.

Figure 13 shows completion times for experiments run with and without W-TPR. As the figure shows, the measurements taken in the absence of W-TPR are only slightly lower than the measurements taken in the presence of W-TPR. We observe that W-TPR incurs relatively little overhead while providing the properties discussed in Section 5.

7. Conclusions and Future Work

We present a real-time scheduling algorithm called HUA and two protocols called D-TPR and W-TPR. The algorithm/protocols’ consider distributable threads and their exception handlers with TUF time constraints. We show that HUA and D-TPR/W-TPR bound the orphan cleanup and recovery time with bounded loss of the best-effort property — the first such algorithm/protocols for systems with (permanent/transient) network failures and unreliable transport. Our implementation using the emerging DRTSJ/RI demonstrates the algorithm/protocols’ effectiveness.

Directions for future work include allowing threads to share non-CPU resources, establishing assurances on thread time constraint satisfactions’, and extending results to arbitrary graph-shaped, multi-node, causal control/data flows.

References