Assured-Timeliness Integrity Protocols for Distributable Real-Time Threads with in Dynamic Distributed Systems

Binoy Ravindran[1], Edward Curley[1], Jonathan S. Anderson[1], and E. Douglas Jensen[2]

[1] Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg Virginia, 24061, USA
{binoy, alias, andersoj}@vt.edu

[2] The MITRE Corporation
Bedford, Massachusetts, 01730, USA
jensen@mitre.org

Abstract

Networked embedded systems present challenges for designers composing distributed applications with dynamic, real-time, and resilience requirements. 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 failure prevents timely execution, the thread must be terminated, requiring detecting and aborting thread orphans and delivering exceptions to the farthest, contiguous surviving thread segment for possible resumption, while optimizing system-wide timeliness. A scheduling algorithm (HUA) and two thread integrity protocols (D-TPR and W-TPR) are presented and shown to bound orphan cleanup and recovery times with bounded loss of best-effort behavior. Implementation experience using the emerging Distributed Real-Time Specification for Java (DRTSJ) 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 [1]. 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 infrastructure , having dynamic node membership and network topology changes, including mobile, ad hoc wireless networks [2].

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 [3] and later in MK7.3 [4], OMG's Real-Time CORBA 1.2 [5], and Sun's emerging Distributed Real-Time Specification for Java (DRTSJ) [6] provide such a model as first-class programming and scheduling abstractions. A distributable thread (see Figure 1) is a thread of execution with a globally unique identity that extends and retracts through local and remote objects, carrying its execution context (e.g., scheduling parameters) as it transits node boundaries [5]. This context is used in resolving resource contention among threads with the objective of maximizing a particular scheduling objective. We focus on distributable threads (hereafter, simply *threads*) as our end-to-end control flow/scheduling abstraction.

During overload it is impossible to meet time constraints for all threads: the demand exceeds the supply. A distinction must be made between urgency and importance in order to select which activities to execute and when (During underloads, such a distinction generally need not be made—e.g., if all time constraints are deadlines, then EDF [7] can meet all deadlines, and no selection must be made.) Traditional deadlines do not capture this distinction, thus we consider the *time/utility function* (or TUF) model [8] that specifies the utility of completing a thread as a function of its completion time. In this paper, we specify a deadline as a binary-valued, downward "step" shaped TUF. A thread's TUF decouples its importance (X-axis) and urgency (Y-axis).

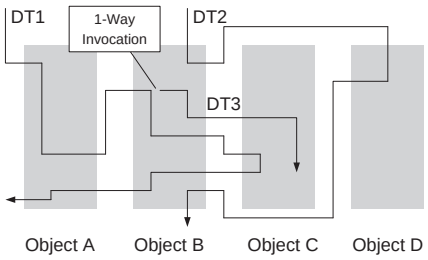


Fig. 1. Distributable Threads

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., [9, 10]).

Our Contributions. When nodes fail, threads may be divided 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.

A straightforward approach for scheduling handlers is to model them as traditional (single-node) threads. Further, the classical *admission control* strategy [11–13] 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. Another distinguishing feature of motivating applications for this model (e.g., [1]) is their relatively long thread execution time magnitudes—e.g., milliseconds to minutes. 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 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.

Thread integrity protocols have been developed in the past—e.g., Thread Polling with bounded Recovery [13], Alpha's Thread Polling [3], Node Alive protocol [14]. None of these efforts provide time-bounded thread cleanup in the presence of node and (permanent/transient) network failures and unreliable transport mechanisms. Further, [13] 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.

2 Models and Objectives

Threads. Threads execute in local and remote objects by location-independent invocations and returns. The portion of a thread executing an object operation is called a *thread segment*; a thread can be viewed as being composed of a series of thread segments. A thread's initial segment is called its *root* and its most recent segment is called its *head*, 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 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). However, the number of thread sections is unknown *a priori*. The application is comprised of a set of threads, denoted $\mathbf{T} = \{T_1, T_2, T_3, \ldots\}$.

Timeliness Model. Each thread T_i's time constraint is specified using a TUF, denoted $U_i\,(t)$. Downward step TUFs 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., [15]).

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.

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, or a thread encounters a network or node failure, an 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.

Each handler may also have a TUF time constraint, and an execution time estimate, provided 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 *nodes* $N_i \in N, i \in [1, m]$ are interconnected via a network. We consider an unreliable multihop network model (e.g., WAN, MANET), with nodes interconnected through routers. Node clocks are synchronized—e.g., using [16].

Nodes may fail arbitrarily by crashing (i.e., fail-stop), while network links may fail transiently or permanently, causing network partitions.

We consider Real-Time CORBA 1.2's [5] *Case 2* approach for thread scheduling. According to this approach, node schedulers use thread scheduling parameters and independently schedule thread sections to optimize the system-wide timeliness optimality criteria, resulting in approximate, global, system-wide timeliness.

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, while retaining the best-effort property of UA algorithms.

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. 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.

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) [10]. On a node, a section's PUD measures the utility that can be accrued by immediately executing it on the node, per unit of remaining execution time.

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.*

In the absence of section failure 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 a failure scenario (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.

Thus, on each node, HUA examines thread sections for potential inclusion in a feasible node schedule in the order of decreasing section PUD. For each

section, the algorithm examines whether that section and its handler can be feasibly completed, in which case it is added to the schedule.

If a non-head section S_i is not included, it 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. 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 included sections always have feasible handlers. Further, all upstream sections' handlers are also feasible. When any such section fails, its handler and all upstream handlers will complete in bounded time.

No such assurances are afforded to sections that fail otherwise—i.e., the termination time expires for 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. S_i's handler is executed in a best-effort manner, in accordance with its potential contribution to the total utility.

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 number of sections of a thread is unknown (otherwise approaches from [17] apply).

A handler is feasible if it can complete before its *absolute* termination time. Failure time is impossible to predict, so 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, delaying the handler's latest start time.

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 introduce a number of variables and auxiliary functions which are largely self-explanatory. Detailed descriptions appear in the full version of this paper.

HUA is shown in Algorithm 1. Invoked at time t_{cur}, HUA H and checks the feasibility of the sections. If a section's earliest predicted completion time exceeds its termination time, it is not included. Otherwise, HUA calculates its PUD. Sections are then sorted by PUD (line 8), and those with positive PUD are iteratively inserted into σ, maintained in the non-decreasing order of termination times. Thus, a section S_i and S_i^h are inserted into positions that correspond to $S_i.X$ and $S_i^h.X$, respectively.

If after inserting S_i and S_i^h, σ becomes infeasible, S_i and S_i^h are removed. If a section S_i is removed and is not a head node, it no longer fits the previous schedule, the integrity protocol is notified regarding S_i's failure. If one

Algorithm 1: HUA: High Level Description

```
1: input: S_r, σ_r, H; output: selected thread S_exe;
2: Initialization: t := t_cur; σ := ∅; HandlerIsMissed := false;
3: updateReleaseHandlerSet ();
4: for each section S_i ∈ S_r do
5:     if feasible(S_i)=false then
6:         reject(S_i);
7:     else S_i.PUD = min( U_i(t+S_i.ExT)/S_i.ExT , U_i^h(t+S_i.ExT+S_i^h.ExT)/(S_i.ExT+S_i^h.ExT) )
8: σ_tmp :=sortByPUD(S_r);
9: for each section S_i ∈ σ_tmp from head to tail do
10:    if S_i.PUD > 0 then
11:        Insert(S_i, σ, S_i.X);
12:        Insert(S_i^h, σ, S_i.X + S_i^h.X);
13:        if feasible(σ)=false then
               Remove(S_i, σ, S_i.X);
14:            Remove(S_i^h, σ, S_i.X + S_i^h.X);
15:            if IsHead(S_i)=false and S_i ∈ σ_r then
16:                alertProtocol(S_i);
17:    else break;
18: if H ≠ ∅ then
19:    for each section S^h ∈ H do
20:        if S^h ∉ σ then
21:            HandlerIsMissed := true;
22:            break;
23: if HandlerIsMissed := true then
24:    S_exe :=headOf(H);
    else
25:    σ_r := σ;
26:    T_exe:=headOf(σ);
27: return S_exe;
```

or more handlers have been released but have not completed their execution , the algorithm checks whether any of those handlers are missing in σ If any handler is missing, the handler at the head of H is selected for execution. If all handlers in H have been included in σ, the section at the head of σ is selected.

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).*[3]

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):

[3] Proofs of all theorems have been eliminted for space, but are available in the full version of this paper at `http://www.real-time.ece.vt.edu/eso07.pdf`.

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; (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, $NBI_{\mathcal{A}}$, 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 [10], LBESA [9], and AUA [13] (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 σ_t is HUA's schedule at time t. DASA's and LBESA's worst-case NBI is zero; AUA's is $+\infty$.*

Theorem 3 *Best-case NBI of HUA, DASA, and LBESA is 0; AUA's is $+\infty$.*

4 The D-TPR Protocol

D-TPR targets systems with node and network failures that are generally permanent. The protocol is instantiated in a per-node component called the Thread Integrity Manager (or TIM), which continually runs D-TPR's polling operation. TIM operations are considered to be administrative operations, and they are conducted with scheduling eligibility exceeding all application threads. We thus ignore the (comparatively small, and bounded) processing delays on each node in the analysis.

4.1 Polling

At every polling interval t_p, the TIM on each node identifies locally-hosted sections, sending a POLL message to each of its predecessor and successor nodes for each section. Each POLL messagecontaining corresponding local and remote section IDs for each section. 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.

4.2 Break Detection

When an invocation is made, D-TPR creates timers which are set to a delay D, the likely worst-case message delay incurred in the network, and is empirically determined (similar to our measurements in Section 6). 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.

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.

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

Lemma 5 *Consider a section S_j and its predecessor S_i. Under D-TPR, if S_i's node fails, or S_j becomes unreachable from S_i (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.*

4.3 Recovery

Recovery operations are administrative functions carried on below the level of application scheduling. While recovery proceeds, D-TPR activities continue concurrently, allowing 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 force the upstream section to become the thread's *new head* while forcing 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 ORPHANPROP message downstream and modifies its downstream POLL messages to include an orphan status. The downstream node will either receive the ORPHANPROP 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 ORPHANPROP message in order to identify all orphans.

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 it's upstream timer will expire, making the section an orphan.

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 directly upstream from a break, D-TPR therefore activates a new head within $t_p + 2D$.*

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 $t_p + 2D$.*

4.4 Cleanup

An *orphan*ed section releases its exception handler only if it is 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+k} to S_j—and will complete by $t_p + (2+k)D + \sigma_{\alpha=0}^{k}(S_{j+\alpha}.X + S_{j+\alpha}^{h}.X)$, unless a section $S_{j+\alpha}$ becomes unreachable from $S_{j+\alpha+1}, 0 \leq \alpha \leq k-1$.*

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

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.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 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 2 shows the section states and transitions in W-TPR. No breaks are ever declared—a section becomes an orphan only if it receives the ORPHAN message from an upstream section. Sections are healthy until notified otherwise.

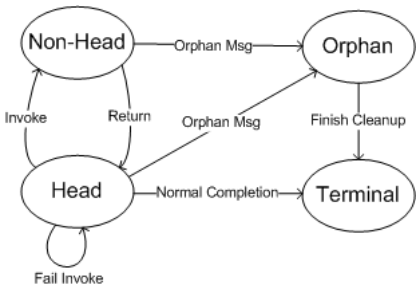


Fig. 2. Section State Diagram

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

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 received by the downstream node, the downstream node attempts to finalize the invocation and sends an INV-ACK message to the upstream section. 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.

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 only a partial invocation was accomplished. Thus the downstream node's INV-ACK/POLL messages are not received upstream while thread execution progresses on the downstream node and further downstream.

Lemma 10 *Under W-TPR thread head location is ambiguous for at most* t_n.

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 and the protocol forces the return message to be resent, which chains upstream. 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 is 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.*

Cleanup. A section becomes an *orphan* upon receipt of an ORPHAN message, in response to its POLL. 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 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}.*

Theorem 12 holds only if no further failures occur between S_j and S_{j+k}. 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.

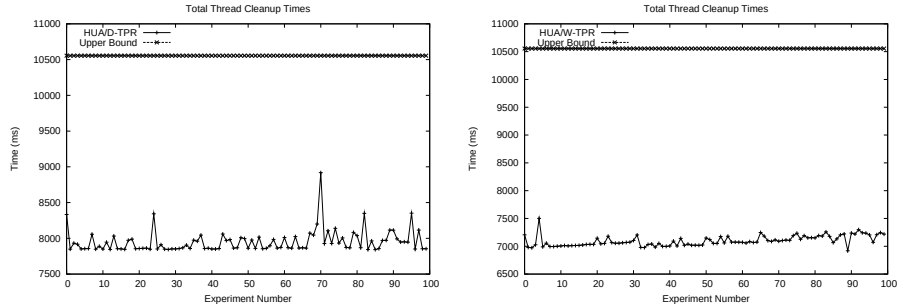
6 Implementation Experience

HUA, D-TPR, and W-TPR were implemented in DRTSJ's RI [6]. The RI includes a threads API, user-space scheduling framework for pluggable thread scheduling, and mechanisms for implementing thread integrity protocols, running atop Apogee's Real-Time Specification for Java (RTSJ)-compliant Aphelion Java Virtual Machine. The experiments and RI ran on the Debian Linux OS (kernel version 2.6.16-2-686) on 800MHz Pentium-III machines.

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. Each experimental run spawned a single distributable thread which propagated to five other nodes, returning back through the same five nodes.

Total 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 3(a) and 3(b) 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. We observe that both HUA/D-TPR and HUA/W-TPR satisfy their cleanup upper bound, validating Theorem 9.

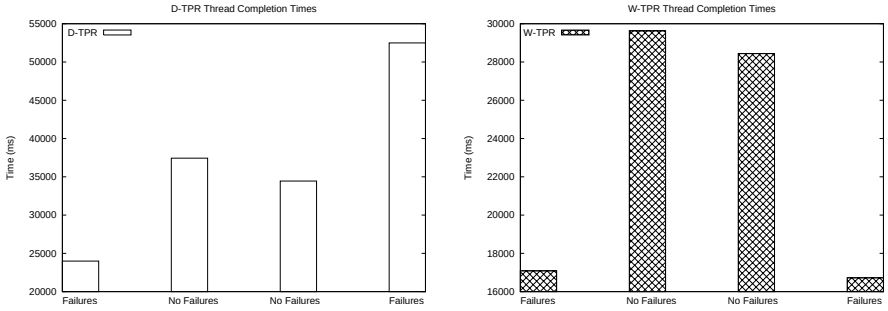
Completion time is the difference between when a root section starts execution and when it completes. Figures 4(a) and 4 show the thread completion times of experiments 1) with failures and D-TPR/W-TPR, 2) without failures but with D-TPR/W-TPR, 3) without failures and without D-TPR/W-TPR, and 4) with failures but without D-TPR/W-TPR. We measure the overhead each protocol incurs in terms of the increase in thread completion times.



(a) D-TPR Thread Cleanup Times (b) W-TPR Thread Cleanup Times

Fig. 3. Thread Cleanup Times for D-TPR and W-TPR

Figure 4(a) 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 4(a) also shows that failed threads with D-TPR complete much more quickly than failed threads with no D-TPR support.



(a) D-TPR Thread Completion Times (b) W-TPR Thread Completion Times

Fig. 4. W-TPR Thread Completion Times

Figure 4 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. 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

1. CCRP: Network centric warfare. `http://www.dodccrp.org/ncwPages/ncwPage.html`
2. Baker, F.: An outsider's view of manet. Internet-Draft, Work In Progress draft-baker-manet-review-01.txt, IETF Network Working Group (March 2002)
3. Northcutt, J.D.: Mechanisms for Reliable Distributed Real-Time Operating Systems — The Alpha Kernel. Academic Press (1987)
4. The Open Group: MK7.3a Release Notes. The Open Group Research Institute, Cambridge, Massachusetts (1998)
5. OMG: Real-time corba 2.0: Dynamic scheduling specification. Technical report, Object Management Group (2001)
6. Anderson, J., Jensen, E.D.: The distributed real-time specification for java: Status report. In: JTRES. (2006)
7. Horn, W.: Some simple scheduling algorithms. Naval Research Logistics Quaterly **21** (1974) 177–185
8. Jensen, E.D., et al.: A time-driven scheduling model for real-time systems. In: IEEE RTSS. (Dec. 1985) 112–122
9. Locke, C.D.: Best-Effort Decision Making for Real-Time Scheduling. PhD thesis, CMU (1986) CMU-CS-86-134.
10. Clark, R.K.: Scheduling Dependent Real-Time Activities. PhD thesis, CMU (1990) CMU-CS-90-155.
11. Nagy, S., Bestavros, A.: Admission control for soft-transactions in accord. In: IEEE RTAS. (1997) 160
12. Streich, H.: Taskpair-scheduling: An approach for dynamic real-time systems. Mini & Microcomputers **17**(2) (1995) 77–83
13. Curley, E., et al.: Recovering from distributable thread failures with assured timeliness in real-time distributed systems. In: IEEE SRDS. (2006) 267–276
14. Goldberg, J., et al.: Adaptive fault-resistant systems (chapter 5: Adpative distributed thread integrity). Technical Report csl-95-02, SRI International (1995)
15. Clark, R., et al.: An adaptive, distributed airborne tracking system. In: IEEE WPDRTS. (April 1999) 353–362
16. Romer, K.: Time synchronization in ad hoc networks. In: ACM MobiHoc. (2001) 173–182
17. Kao, B., et al.: Deadline assignment in a distributed soft real-time system. IEEE TPDS **8**(12) (1997) 1268–1274