Recovering from Distributable Thread Failures in Distributed Real-Time Java

Edward Curley*, Binoy Ravindran*, Jonathan Anderson*, and E. Douglas Jensen‡
*ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{alias,binoy,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 ("threads") in distributed real-time systems that operate under run-time uncertainties including those on thread execution times, thread arrivals, and node failure occurrences. When a thread experiences a node failure, the result is broken thread having an orphan. Under a termination model, the orphans must be detected and aborted, and exceptions must be delivered to the farthest, contiguous surviving thread segment for resuming thread execution. Our application/scheduling model includes the proposed distributable thread programming model for the emerging Distributed Real-Time Specification for Java (DRTSJ), together with an exception handler model. Threads are subject to time/utility function (TUF) time constraints and an utility accrual (UA) optimality criterion. A key underpinning of the TUF/UA scheduling paradigm is the notion of "best-effort" where higher importance threads are always favored over lower importance ones, irrespective of thread urgency as specified by their time constraints. We present a thread scheduling algorithm called HUA and a thread integrity protocol called TPR. We show that HUA and TPR bound the orphan cleanup and recovery time with bounded loss of the best-effort property. Our implementation experience of HUA/TPR in the Reference Implementation of the proposed programming model for the DRTSJ demonstrates the algorithm/protocol's effectiveness.

Index Terms

distributable thread, thread integrity, time/utility function, utility accrual scheduling, distributed real-time Java, scheduling, real-time

I. Introduction

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

An abstraction for programming multi-node sequential behaviors and for enforcing end-to-end properties is *distributable threads* [1], [2]. Distributable threads first appeared in the Alpha OS [2], and later in Mach 3.0 [3] (a subset), and MK7.3 [4]. They constitute the first-class programming and scheduling abstraction for multi-node sequential behaviors in Real-Time CORBA 2 [5] and are proposed for Sun's emerging Distributed Real-Time Specification for Java (DRTSJ) [1]. In the rest of the paper, we will refer to distributable threads as *threads*, unless qualified.

A thread is a single logically distinct (i.e., having a globally unique identity) locus of control flow movement that extends and retracts through local and (potentially) remote objects. The objects in the distributable thread model are passive (as opposed to active objects that encapsulate one or more local threads). An object instance resides on a single computational node. A distributable thread enters an

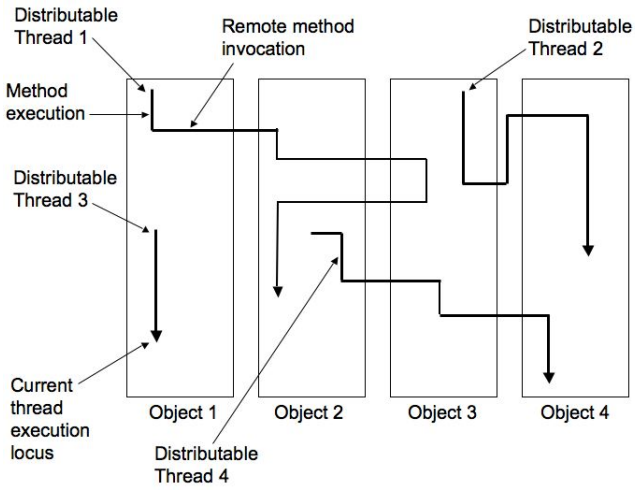

Fig. 1. Distributable Threads

object instance by invoking one of its operations; that portion in the object instance is a *segment* of the distributable thread, and is implemented as a local thread. If a thread invokes a sequence of methods in object instances on the same node, that sequence of segments is called a *section*. Only the *head* segment of a distributed thread is active (executing), all others are suspended (i.e., they have made remote invocations to other nodes, eventually to the head).

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 (i.e., segments) 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 a snapshot in time of four threads (and their segments) prior to when they return from their invocations [5].

Except for the required execution context, the abstraction imposes no constraints on the presence, size, or structure of any other data that may be propagated as part of the thread's flow. Commonly, input parameters may be propagated with thread invocations, and results may be propagated back with returns. When movement of data associated with a thread is the principal purpose for a thread, the abstraction can be viewed as a data flow one as much as, or more than, a control flow one. Whether an instance of the abstraction is regarded as being an execution flow one or a data flow one, the invariants are that: the (pertinent portion of the) application is structured as causal linear sequence of invocations from one object to the next, unwinding back to the initial point; each invoked object's ID is known by the invoking object; and there are end-to-end properties that must be maintained, including timeliness, thread fault management, and thread control (e.g., concurrency, pause/resume, signaling of state changes).

In this paper, we consider threads as the programming and scheduling abstraction for dynamic distributed real-time systems that operate under run-time uncertainties. These uncertainties include arbitrary node failures, unbounded thread execution time behaviors (due to context-dependence), and arbitrary thread arrival behaviors. The uncertainties on thread execution time and arrival behaviors can cause transient and sustained resource overloads.

When overloads occur, optimally satisfying the time constraints of all threads (e.g., meeting all their deadlines) is impossible as the computational demand exceeds the supply. The urgency (i.e., time-criticality) 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 thread 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 relative importance during overloads.

(During underloads, such a distinction generally need not be made, especially if all time constraints are deadlines, as optimal algorithms exist that can meet all deadlines—e.g., EDF [6].)

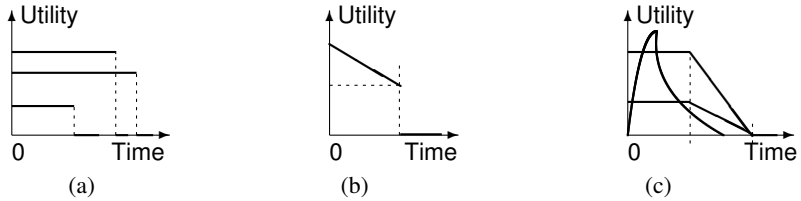

Fig. 2. Example TUF Time Constraints: (a) Step TUFs; (b) TUF of an AWACS [7]; and (c) TUFs of a Coastal Air defense System [8].

Deadlines cannot express both urgency and importance. Thus, we employ the *time/utility function* (or TUF) timeliness model [9] that specifies the utility of completing a thread as a function of that thread's completion time.

The *utility* of an entity (e.g., thread) contending for a sequentially shared resource is an abstract value that is an application-specific function of when that entity completes its use of the resource after it has been granted access. Thus, a deadline time constraint is a binary-valued, downward "step" shaped TUF; Figure 2(a) shows examples. A thread's TUF decouples its importance and urgency—urgency is measured on the X-axis, and we denote importance with utility on the Y-axis.

Some real-time systems also have threads with *non-deadline* time constraints, such as those where the utility attained for thread completion *varies* (e.g., decreases, increases) with completion time. Figures 2(b)–2(c) show example TUFs from two notional defense applications [7], [8].

Non-time-constrained threads have TUF's with constant values signifying their relative importances; this allows both time-constrained and non-time-constrained threads to be scheduled with the same UA algorithm.

When thread time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued thread utility—e.g., maximizing the sum of the threads' utilities. Such criteria are called *utility accrual* (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that optimize UA criteria are called UA sequencing algorithms (see [10] for examples).

UA algorithms that maximize total utility under downward step TUFs (e.g., [11], [12]) default to EDF during underloads, since EDF satisfies all deadlines during underloads. Consequently, they obtain the optimum total utility during underloads. During overloads, they inherently favor more important threads over less important ones (since more utility can be attained from the former), irrespective of thread urgency, and thus exhibit adaptive behavior and graceful timeliness degradation. This behavior of UA algorithms is called "best-effort" [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.[1] Consequently, high importance threads that arrive at any time always have a very high likelihood for feasible completion (irrespective of their urgency). Note that EDF's optimal timeliness behavior is a special-case of UA scheduling.

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

When nodes transited by threads fail, this can cause threads that span the nodes to break by dividing them into several pieces. Segments of a thread that are disconnected from its node of origin (called the thread's *root*), are called *orphans*. When threads experience failures causing orphans, application-supplied exception handlers must be released for execution on the orphans' nodes. Such handlers may have time constraints themselves and will compete for their nodes' processor along with 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

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

handlers complete their execution, the application may desire to resume the execution of the failed thread from the contiguous surviving thread segment farthest 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-cleanup handlers along with threads must contribute to system-wide timeliness optimality. Untimely handler execution can degrade timeliness optimality—e.g.: high urgency handlers are delayed by low urgency non-failed threads, thereby delaying the resumption of high urgency failed threads; high urgency, non-failed threads are delayed by low urgency handlers.

A straightforward approach for scheduling handlers is to conceptually model them as traditional (single-node) threads, insert them into the ready queue when distributable threads arrive at nodes, and schedule them along with the threads on those nodes, according to a discipline that provides acceptable system-wide timeliness. This should be possible, as handlers are like single-node threads, with similar scheduling parameters (e.g., execution time, time constraints). However, constructing a schedule that includes a thread and its handler on a node implies that the thread *and* the handler will be dispatched for execution according to their order in the schedule. This is not true, as the handler needs to be dispatched only if and when the thread fails at an upstream node causing an orphan on the handler's node. Furthermore, when a thread is released for execution, which is a scheduling event[2], it is immediately ready for execution. However, its handler is released for execution only if and when the thread fails at an upstream node. Thus, constructing a schedule at a thread's release time on a node such that it also includes the thread's handler on the node will require a prediction of when the handler will be ready for execution in the future — a potentially impossible problem as there is no way to know if a thread will fail.

These problems can possibly be alleviated by considering a thread's failure time as a scheduling event and constructing schedules on the thread's orphan nodes that include the handlers at that time (e.g., as in [2]). However, this would mean that there is no way to know whether or not the handlers can feasibly complete until the thread fails. In fact, it is possible that when the thread fails, the schedulers on handlers' nodes' may discover that the handlers are infeasible due to node overloads — e.g., there are more threads on those nodes than can be feasibly scheduled, and there exists schedules of threads (on those nodes) excluding the handlers from which more utility can be attained than from ones including the handlers.

Another strategy that avoids this predicament and has been very often considered in the past (e.g., [13]–[15]) is classical *admission control*: When a thread arrives at a node, check whether a feasible schedule can be constructed on that node that includes all the previously admitted threads and their handlers, besides the newly arrived one and its handler. If so, admit the thread and its handler; otherwise, reject. But this will cause the very fundamental problem that is solved by UA schedulers through their best-effort decision making—i.e., a newly arriving thread is rejected because it is infeasible, despite that thread being more important than the other threads on that node. 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 one.

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

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

We present a UA scheduling algorithm called *Handler-assured Utility Accrual scheduling algorithm* (or HUA) for thread scheduling, and a protocol called *Thread Polling with bounded Recovery* (or TPR) for ensuring thread integrity. We show that HUA in conjunction with TPR ensures that handlers of threads that encounter failures during their execution will complete within a bounded time, yielding bounded thread

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

cleanup time. Yet, the algorithm/protocol retains the fundamental best-effort property of UA algorithms with bounded loss—i.e., a high importance thread that may arrive at any time has a very high likelihood for feasible completion. Our implementation experience of HUA/TPR using the Reference Implementation of the proposed DRTSJ demonstrates the algorithm/protocol's effectiveness.

Similar to UA algorithms, integrity protocols for threads have been developed in the past—e.g., Alpha's Thread Polling protocol [2], the Node Alive protocol [16], and adaptive versions of Node Alive [16]. However, none of these protocols in conjunction with a scheduling algorithm provide time-bounded thread cleanup. Our work builds upon our prior work in [15] that provides bounded thread cleanup. However, [15] does so through admission control and thus suffers from unbounded loss of the best-effort property (we show this in Section IV-G). In contrast, HUA/TPR provides bounded thread cleanup with bounded loss of the best-effort property. Thus, the paper's contribution is the HUA/TPR algorithm/protocol.

The rest of the paper is organized as follows: In Section II, we discuss the motivating application context. Section III describes the models and algorithm/protocol objectives. Section IV presents HUA and Section V presents TPR. In Section VI, we discuss our implementation experience. In Section VII, we review past and related efforts, and contrast them with our work. We conclude the paper in Section VIII.

II. Motivating Application Context

An example distributed real-time system application context that motivates our work is the U.S. DoD's information age warfare transformation vision called Network Centric Warfare (NCW), which focuses on getting the right information to the right entity (task, person, etc.) at the right time, on and beyond the battlefield [17]. Our motivating NCW scenario is a single integrated non-hierarchically distributed air defense network that consists of a set of combat and surveillance platforms (e.g., ships, aircraft) with components for managing on-board sensors, weapons, tracking systems, and battle management/command and control (BM/C2) operations. A ship may detect a threat in the airspace, or be warned of it by another ship, an aircraft, or a satellite. The threatened ship may be unable to prosecute the threat (by launching a weapon) due to current limitations of its weapon systems (e.g., limited sensor range for weapon guidance or lack of suitable weapon). Consequently, prosecution of the threat may be assigned to a platform in the vicinity (e.g., another ship, or an aircraft), which may launch a weapon. Yet another platform in the vicinity may guide the weapon to the target until the engagement is complete. The pattern of interactions is peer-to-peer, departing from the hierarchical interaction pattern that has dominated traditional BM/C2 operations. Distributed (i.e., multinode) activities in this scenario include those for threat detection, identification, tracking, weapons assignment, and weapon guidance. These activities have time constraints, and the most important ones must be satisfied acceptably well (e.g., it may be acceptable for a weapon to detonate close enough to damage its target), despite node overloads and failures, to achieve acceptable mission measures of effectiveness.

A distinguishing feature of this application context is the relatively long magnitudes of activity execution and system reaction times, compared to those of traditional real-time subsystems—e.g., seconds to minutes. Such longer time magnitudes present opportunities for highly effective dynamic real-time scheduling and timeliness optimization with costs that are relatively larger than those of traditional real-time scheduling disciplines.

III. Models and Objectives

A. Distributable Thread Abstraction

Threads execute in local and remote object instances 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. Further details of the thread model can be found in [1], [2], [5].

Execution time estimates of the sections of a thread are assumed to be known. This time estimate is not the worst-case; it can be violated at run-time (e.g., due to context dependence) and can cause processor overloads. A section's time estimate is presented by a thread (to the node scheduler) when the thread arrives at the node—e.g., the time estimate may be propagated with the thread; it may be presented by a portable interceptor at the node.

The total number of sections of a thread is assumed to be unknown a-priori, as a thread is assumed to make remote invocations and returns based on context-dependent application logic.

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

B. Timeliness Model

Usually run-times such as operating systems designate an entire thread for the duration of its existence as being either a real-time one or a non-real-time one. Instead, our model provides a finer granularity, and often more useful, designation. A thread becomes a real-time (i.e., time-constrained) one while it has a completion time constraint (such as a deadline); a special case is the conventional one that a thread is designated as a real-time one when it is created and for the duration of its existence. In our model, a time constraint is imposed on a thread by a declaration in the code such as Real-Time CORBA's `begin_scheduling_segment(TC(i),DL, 500mS)` [5] that is a system call to the scheduler. In this example, when the thread executes that system call, the run-time (e.g., OS) begins a count-down timer associated with that thread, corresponding to the 500 mS deadline having the ID `TC(i)`. A matching declaration later in the code such as `end_scheduling_segment(TC(i))` signifies the end of the time constraint (e.g., deadline) `TC(i)`.

The thread must execute this system call before the 500 mS deadline. If it does, the run-time cancels the timer and the thread has satisfied the time constraint, and from that point it becomes a non-real-time thread unless or until it encounters a subsequent time constraint declaration. If it does not, the thread has failed to satisfy the time constraint, the run-time's corresponding timer times out, and an exception occurs. This lexically scoped `begin/end` pair of system calls is called a *scheduling segment scope.* "Segment" refers to the fact that distributable thread segments (implemented as local threads) are the locally scheduled entity.

Scheduling segments may span processor boundaries—i.e., a thread may enter a scheduling segment at a node, leave the node via a remote invocation, return to the node after the invocation, and then exit the scheduling segment.

We specify the time constraint of each time-constrained thread using a TUF. The TUF of a thread T_i is denoted as $U_i(t)$. Thus, thread T_i's completion at a time t will yield an utility $U_i(t)$. Though TUFs can take arbitrary shapes, here we focus on *non-increasing* unimodal TUFs, as they encompass the majority of the time constraints of interest to us. Figures 2(a), 2(b), and two TUFs in Figure 2(c) show examples from some of our experiments. (Unimodal TUFs are those for which any decrease in utility cannot be followed by an increase.) A thread presents its TUF to the scheduler through the API that it uses to enter the corresponding scheduling segment (e.g., `begin_scheduling_segment`).

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. Also, $U_i(t) > 0, \forall t \in [I_i, X_i]$ and $U_i(t) = 0, \forall t \notin [I_i, X_i], i \in [1, n]$.

C. Exceptions and Abort Model

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

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

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

Once all handlers complete their execution, TPR delivers a failure-exception notification to the contiguous surviving thread segment farthest from the thread's root. This is the segment from where thread execution can potentially be resumed in an application-specific manner—e.g., the thread may unwind back; the thread may make alternative remote invocations, etc. By delivering the failure-exception notification to the farthest, contiguous surviving thread segment, the thread integrity mechanism is able to preserve the abstraction of a continuous reliable distributable thread.

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

A handler also specifies 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 section's held resources and return the system to a safe state.

D. Resource Model

Thread sections can access non-processor resources (e.g., disks, network interface controllers) located at their nodes during their execution. Such resources, in general, are serially reusable, and can be shared under mutual exclusion constraints. (Exception handlers of sections, however, are not allowed to mutually exclusively access resources.) Similar to fixed-priority resource access protocols [18] and that for TUF algorithms [12], [19], we consider a single-unit resource model.

A thread may request multiple shared resources during its lifetime. The requested time intervals for holding the resources may be nested, overlapped, or disjoint. Threads explicitly release all granted resources before the end of their executions.

All resource request/release pairs are assumed to be confined within nodes. Thus, a thread cannot lock a resource on one node and release it on another node. Note that once a thread locks a resource on a node, it can make remote invocations (carrying the lock with it). Since request/release pairs are confined within nodes, the lock is released after the thread's head returns back to the node where the lock was acquired.

Threads are assumed to access resources arbitrarily—i.e., which resources will be needed by which threads, and in what order are not a-priori known. Consequently, we consider a deadlock detection and resolution strategy (as opposed to deadlock avoidance or prevention). A deadlock is resolved by aborting a thread involved in the deadlock, by executing the thread's handler (which will perform the necessary resource roll-backs/roll-forwards).

E. System and Failure Models

We consider a system model where a set of processing components, generically referred to as *nodes*, are interconnected via a network. Each node executes thread sections. The order of executing sections on a node is determined by the scheduler residing at the node.

We consider the *Case 2* approach of Real-Time CORBA for thread scheduling. According to this approach, node schedulers use the propagated thread scheduling parameters and independently schedule

thread sections on their respective nodes using the same timeliness optimality criterion. Thus, scheduling decisions made by a node scheduler are independent of those of other node schedulers. Though this results in approximate, global, system-wide timeliness, Real-Time CORBA explicitly supports the approach, due to its simplicity, message-efficiency (from the thread scheduling standpoint), and capability for coherent end-to-end scheduling. Real-Time CORBA also describes — but does not support — Cases 1, 3, and 4, which describe non real-time, global, and multilevel distributed scheduling, respectively. The Case 3 approach, in contrast with Case 2, advocates distributed scheduling, where nodes explicitly cooperate to construct system-wide thread schedules, incurring message overhead costs. An example scheduling algorithm that follows the Case 3 paradigm is described in [20]. We follow the Case 2 approach.

We consider a single hop network model (e.g., a local area network), with nodes interconnected through a hub or a switch. We assume a reliable message transport protocol with a worst case message delivery latency D (as opposed to the case 3 message costs).

We denote the set of nodes as $N_i \in N, i \in [1, m]$. We assume that all node clocks are synchronized using a protocol such as [21]. We consider an arbitrary, crash failure model for the nodes.

F. Scheduling Objectives

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

IV. The HUA Algorithm

A. Rationale

Section Scheduling. Since the task model is dynamic—i.e., when threads will arrive at nodes, how many sections a thread will have, which set of resources will be needed by which threads, the length of time for which those resources will be needed, and the order of accessing the resources are all statically unknown, future scheduling events (e.g., new thread arrivals, resource requests, time constraint changes, thread execution time changes) cannot be considered at a scheduling event. Thus, section schedules must be constructed on the system nodes by solely exploiting the current system knowledge.

Since the primary scheduling objective is to maximize the total thread accrued utility, a reasonable heuristic is a "greedy" strategy at each node: favor "high return" thread sections over low return ones, and complete as many of them as possible before thread termination times, as early as possible (since TUFs considered here are non-increasing).

The potential utility that can be accrued by executing a section on a node defines a measure of that section's "return on investment." We measure this using a metric called the *Potential Utility Density* (or PUD) originally introduced in [12]. On a node, a section's PUD measures the utility that can be accrued per unit time by executing the section and those section(s) that it (directly or transitively) depends upon for locked resources.

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

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

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

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

Since a section's best-case failure scenario is the absence of a failure for the section and all of its dependents, the corresponding section PUD can be obtained as the total utility accrued by executing the

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

Thus, on each node, HUA examines sections for potential inclusion in a feasible schedule for the node in the order of decreasing section PUDs. For each section, the algorithm examines whether that section and its handler, along with the section's dependents and their handlers, can be feasibly completed (we discuss section and handler feasibility later in this subsection). If infeasible, the section, its handler, the section's dependents, and their handlers are deferred. 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 deferred can be the head of S_i's thread T_i; if so, S_i is reconsidered for scheduling at subsequent scheduling events on S_i's node, say N_i, until T_i's termination time expires.

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

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

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

Feasibility. Feasibility of a section on a node can be tested by verifying whether the section can be completed on the node before the section's distributable thread's end-to-end termination time. Using a thread's end-to-end termination time for verifying the feasibility of a section of the thread may potentially overestimate the section's slack, especially if there are a significant number of sections that follow it in the thread. However, this is a reasonable choice, since we do not know the total number of sections of a thread. If the total number of sections of a thread is known a-priori, then schemes such as [22] that distribute the thread's total slack (equally, or proportionally) among all its sections can be considered.

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

There is always the possibility that a new section S_i is released on a node after the failure of another section S_j at the node (per Definition 1) and before the completion of S_j's handler on the node. As per the best-effort philosophy, S_i must immediately be afforded the opportunity for feasible execution on

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

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 deferred now, as that will violate the commitment previously made to S_j, the only option left is to not consider S_i for execution until S_j's handler completes, consequently degrading the algorithm's best-effort property. In Section IV-G, we quantify this loss.

B. Algorithm Overview

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

- $\mathcal{S}_r$ is the current set of unscheduled sections including a newly arrived section (if any). $S_i \in \mathcal{S}_r$ is a section. S_i^h denotes S_i's handler. T_i denotes the thread to which a section S_i and S_i^h belong.
- σ_r is the schedule (ordered list) constructed at the previous scheduling event. σ is the new schedule.
- $U_i(t)$ denotes S_i's TUF, which is the same as that of T_i's TUF. $U_i^h(t)$ denotes S_i^h's TUF.
- $S_i.X$ is S_i's termination time, which equals T_i's termination time. $S_i.ExecTime$ is S_i's estimated remaining execution time. $S_i.Dep$ is S_i's dependency list.
- 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.
- `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.
- `Owner(`R`)` denotes the sections that are currently holding resource R.
- `reqRes(`T`)` returns the resource requested by section (or thread) T. If T is not requesting any resource, function returns $\emptyset$. If T has made a remote invocation and has not returned from it, function returns the keyword $REMOTE$.
- `notifyTPR(`S_i`)` declares S_i as failed (by not sending a SEG_ACK message for S_i in response to the ROOT_ANNOUNCE broadcast message of the TPR protocol — see Section V for TPR details).
- `IsHead(`S`)` returns true if S is a head; false otherwise.
- `headOf(`σ`)` returns the first section in σ.
- `sortByPUD(`σ`)` returns a schedule ordered by non-increasing section PUDs. If two or more sections have the same PUD, the section(s) with the largest $ExecTime$ will appear before any others with the same PUD.
- `Insert(`S`,`σ`,`I`)` inserts section S in the ordered list σ at the position indicated by index I; if entries in σ exists with the index I, S is inserted before them. After insertion, S's index in σ is I.
- `Remove(`S`,`σ`,`I`)` removes section S from ordered list σ at the position indicated by index I; if S is not present at the position in σ, the function takes no action.
- `lookup(`S`,`σ`)` returns the index value of the first occurrence of S in the ordered list σ.
- `feasible(`σ`)` returns a boolean value indicating schedule σ's feasibility. σ is feasible, if the predicted completion time of each section S in σ, denoted $S.C$, does not exceed S's termination time. $S.C$ is the time at which the scheduler is invoked plus the sum of the $ExecTime$'s of all sections that occur before S in σ and $S.ExecTime$.

Algorithm 1 describes HUA at a high level of abstraction. When invoked at time t_{cur}, HUA first updates the set H (line 3) and checks the feasibility of the sections. If a section's earliest predicted completion time exceeds its termination time, it is (at least temporarily) rejected (line 6). Otherwise, HUA calculates the section's *Local Utility Density* (or LUD) (lines 7-9) as the minimum of the PUDs for the section's best-case and worst-case failure scenarios, and builds its dependency list (line 10).

The PUD of each section is computed by the procedure `calculatePUD()`, and the sections are then sorted by their PUDs (lines 11–13).

```
1:  input: S_r, σ_r, H; output: selected section 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);
        else
7:          PUD_B = U_i(t+S_i.ExecTime) / S_i.ExecTime ;
8:          PUD_W = U_i^h(t+S_i.ExecTime+S_i^h.ExecTime) / (S_i.ExecTime+S_i^h.ExecTime) ;
9:          S_i.LUD = min(PUD_B, PUD_W) ;
10:         S_i.Dep := buildDep(S_i);
11: for each section S_i ∈ S_r do
12:     S_i.PUD:=calculatePUD(S_i, t);
13: σ_tmp :=sortByPUD(S_r);
14: for each section S_i ∈ σ_tmp from head to tail do
15:     if S_i.PUD > 0 then
16:         σ := insertByETF(σ, S_i, σ_r);
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:     S_exe:=headOf(σ);
27: return S_exe;
```

Algorithm 1: HUA: High Level Description

In each step of the *for*-loop from line 14 to 17, the section with the largest PUD, its handler, the section's dependents, and their handlers are inserted into σ, if it can produce a positive PUD. The output schedule σ is then sorted in the non-decreasing order of section termination times by the procedure `insertByETF()`.

If one or more handlers have been released but have not completed their execution (i.e., $H \neq \emptyset$; line 18), HUA checks whether any of those handlers are missing in the schedule σ (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 σ, the section at the head of σ is selected (line 26).

It is possible for the thread of the section that is at the head of σ to be blocked on a remote invocation (see Section IV-C). If that happens, then no section is dispatched for execution, until the next scheduling event.

C. Computing Dependency Lists

HUA builds the *dependency list* of each section—that arises due to mutually exclusive resource sharing—by following the chain of resource request and ownership.

```
1:  input: Section S_k; output: S_k.Dep ;
2:  Initialization : S_k.Dep := S_k; Prev := S_k;
3:  while (reqRes(Prev) ≠ ∅ ⋀
        reqRes(Prev) ≠ 'REMOTE' ⋀
        Owner(reqRes(Prev)) ≠ ∅ ) do
4:      S_k.Dep :=Owner(reqRes(Prev)) ·S_k.Dep;
5:      Prev := Owner(reqRes(Prev));
```

Algorithm 2: `buildDep(`S_k`)`: Building Dependency List for a Section S_k

Algorithm 2 shows this procedure for a section S_k. For convenience, the section S_k is also included in its own dependency list. Each section S_l other than S_k in the dependency list has a successor section that needs a resource which is currently held by S_l. Algorithm 2 stops either because: (1) a predecessor section does not need any resource; or (2) the requested resource is free; or (3) a predecessor section's thread has made a remote invocation and has not returned from it. The last case occurs due to remote dependencies—e.g., a section S_i is blocked on a resource that is held by a section S_j, and S_j's thread has made a remote invocation and is yet to return from it (for this case, the function `reqRes()` returns the keyword $REMOTE$). Note that we use the notation "**.**" to denote an append operation. Thus, the dependency list starts with S_k's farthest predecessor and ends with S_k.

D. Resource and Deadlock Handling

Two kinds of deadlock can occur among distributable threads: a) local deadlocks—i.e., two thread sections on the same node become blocked on each other due to locks that are held by the other, and b) distributed deadlocks—i.e., two threads on two different nodes become blocked on each other due to remotely held locks.

```
1:  input: Requesting section S_k, t_cur;
    /* deadlock detection */;
2:  Deadlock := false;
3:  S_l := Owner(reqRes(S_k));
4:  while S_l ≠ ∅ do
5:      S_l.LUD := U_{S_l}(t_cur + S_l.C)/S_l.C;
6:      S_l.LUD = min( U_l(t_cur+S_l.ExecTime)/S_l.ExecTime , U_l^h(t_cur+S_l.ExecTime+S_l^h.ExecTime)/(S_l.ExecTime+S_l^h.ExecTime) );
7:      if S_l = S_k then
8:          Deadlock := true;
9:          break;
        else
10:         S_l := Owner(reqRes(S_l));
    /* deadlock resolution if any */;
11: if Deadlock = true then
12:     abort(The section S_m with the lowest LUD in the cycle);
```

Algorithm 3: Deadlock Detection and Resolution

To handle local deadlocks, we consider a deadlock detection and resolution strategy, instead of a deadlock prevention or avoidance strategy, due to the dynamic nature of the systems of interest — which resources will be needed by which sections, for how long, and in what order, are all unknown to the scheduler. Under a single-unit resource request model, the presence of a cycle in the resource graph is the necessary and sufficient condition for a deadlock to occur. Thus, a deadlock can be detected by a straightforward cycle-detection algorithm. Such an algorithm is invoked by the scheduler whenever a section requests a resource. A deadlock is detected if the new edge resulting from the section's resource request produces a cycle in the resource graph. To resolve the deadlock, some section needs to be aborted, which will result in some utility loss. To minimize this loss, we compute the utility that a section can potentially accrue by itself if it were to continue its execution, which is measured by its LUD (line 9, Algorithm 1). HUA aborts that section in the cycle with the lowest LUD. Algorithm 3 describes this procedure.

Timeliness of the system can be improved if we preempt a section S instead of aborting it, given that S can complete before its termination time. We can roll-back and add the section into the unordered schedule at the next scheduling event. To roll-back S, at each resource request, a checkpoint should be saved for it, since resource requests are the events causing deadlocks. This can be a future improvement of HUA.

Detection and resolution of distributed deadlocks require node schedulers to cooperate for constructing thread section schedules, and hence is beyond the scope of Real-Time CORBA's Case 2 approach per se (where node schedulers independently make scheduling decisions) and this paper. Thus, under HUA, distributed deadlocks will be detected by the affected node schedulers when the termination times of the deadlocked threads eventually expire, triggering the immediate release of the section abort handlers on those nodes. This deadlock detection/resolution approach is a tradeoff of the Case 2 approach versus other approaches, especially distributed scheduling approaches (e.g., Case 3), where node schedulers cooperate to construct thread schedules, detecting and resolving distributed deadlocks during that process, at the expense of increased scheduling cost—i.e., a tradeoff of distributed deadlock-optimistic, low cost scheduling versus distributed deadlocks-free, high cost scheduling.

E. Computing Section PUD

Procedure `calculatePUD()` (Algorithm 4) accepts a section S_i (with its dependency list) and the current time t_{cur}. It determines S_i's PUD, by assuming that sections in $S_i.Dep$ and their handlers are executed from the current position (at t_{cur}) in the schedule, while following the dependencies.

1: **input**: S_i, t_{cur}; **output**: $S_i.PUD$;
2: *Initialization* : $t_c := 0$, $t_c^h := 0$, $U := 0$, $U^h := 0$;
3: **for** *each section* $S_j \in S_i.Dep$, *from tail to head* **do**
4: $\quad t_c := t_c + S_j.ExecTime$;
5: $\quad U := U + U_j(t_{cur} + t_c)$;
6: $\quad t_c^h := t_c^h + S_j^h.ExecTime$;
7: $\quad U^h := U^h + U_j^h(t_{cur} + t_c + t_c^h)$;
8: $S_i.PUD := \min\left(U/t_c, U^h/(t_c + t_c^h)\right)$;
9: **return** $S_i.PUD$;

Algorithm 4: `calculatePUD(`S_i, t_{cur}`)`: Calculating the PUD of a Section S_i

To compute S_i's PUD at time t_{cur}, HUA computes the PUDs for the best-case and worst-case failure scenarios and determines the minimum of the two.

For determining S_i's total accrued utility for the best-case failure scenario, HUA considers each section S_j that is in S_i's dependency chain, which needs to be completed before executing S_i. The total expected execution time upon completing S_j is counted using the variable t_c of line 4 (Algorithm 4). With the known expected completion time of each section, we can derive the expected utility for each section, and thus obtain the total accrued utility U (line 5) for T_i's best-case failure scenario.

For determining S_i's total accrued utility for the worst-case failure scenario, the algorithm counts the total expected execution time upon completing T_j's handler using the variable t_c^h of line 6. The total accrued utility for the worst-case failure scenario U^h can be determined once the section's completion time followed by its handler's completion time is known (line 7).

The best-case and worst-case failure scenario PUDs can be determined by dividing U and U^h by t_c and $t_c + t_c^h$, respectively, and the minimum of the two PUDs is determined as S_i's PUD (line 8).

Note that the total execution time of S_i and its dependents consists of: (1) the time needed to execute the sections that directly or transitively block S_i; and (2) S_i's remaining execution time. By `buildDep()`'s operation, all the dependent sections are included in $S_i.Dep$.

Note also that each section's PUD is calculated assuming that it is executed at the current position in the schedule. This would not be true in the output schedule σ, and thus affects the accuracy of the PUDs calculated. We are calculating the highest possible PUD of each section by assuming that it is executed at the current position in the schedule, and that sections release resources only after their executions complete. Intuitively, this would benefit the final PUD, since `insertByETF()` always selects the section with the highest PUD at each insertion on σ. Also, the PUD calculated for the dispatched section at the head of σ is always accurate.

Thus, the PUD calculation for a section reflects the fact that executing the sequence of the section, its dependents, and the handlers will require a total time equal to the sum of the individual execution times and will yield a total utility equal to the sum of the individual utilities. A section's PUD, thus measures the section's "return on investment." Note that the term "potential" is used in PUD. This is because the PUD calculation measures the (highest possible) utility that can possibly be obtained from the aggregate computation (of a section, its dependents, and handlers), given the current system knowledge. It is quite possible that future situations (e.g., new dependencies, execution overruns) may negate the chance to accrue any utility from the aggregate computation.

F. Constructing Termination Time-Ordered Schedules

Algorithm 5 describes `insertByETF()` (invoked in Algorithm 1, line 16). `insertByETF` updates the tentative schedule σ by attempting to insert each section, along with its handler, all of the section's dependent sections, and their handlers into σ. The updated schedule σ is an ordered list of sections, where each section is placed according to the termination time that it *should* meet.

```
 1:  input  : S_i, an ordered section list (schedule) σ, and schedule at the previous scheduling event σ_r
 2:  output : the updated list σ
 3: if S_i ∉ σ then
 4:     Copy σ into σ_tmp: σ_tmp :=σ;
 5:     Insert(S_i, σ_tmp, S_i.X);
 6:     Insert(S_i^h, σ_tmp, S_i.X + S_i^h.X);
 7:     CuTT = S_i.X;
 8:     for each section S_j ∈ {S_i.Dep − S_i} from head to tail do
 9:         if S_j ∈ σ_tmp then
10:             TT=lookup(S_j, σ_tmp);
11:             if TT < CuTT then
                    continue;
12:             else
13:                 Remove(S_j, σ_tmp, TT);
14:                 TTh=lookup(S_j^h, σ_tmp);
15:                 Remove(S_j^h, σ_tmp, TTh);
16:         CuTT:=min(CuTT, S_j.X);
17:         Insert(S_j, σ_tmp, CuTT);
18:         Insert(S_j^h, σ_tmp, S_j.X + S_j^h.X);
19:     if feasible(σ_tmp) = true then
20:         σ := σ_tmp;
21:     else
22:         if IsHead(S_i) =false and S_i ∈ σ_r then
23:             notifyTPR(S_i);
24: return σ;
```

Algorithm 5: `insertByETF` (σ, S_i, σ_r) : Inserting a Section S_i, S_i's Handler, S_i's Dependents, and their Handlers into a Feasible Schedule σ

Note that the time constraint that a section should meet is not necessarily its termination time. In fact, the index value of each section in σ is the actual time constraint that the section should meet.

A section may need to meet an earlier termination time in order to enable another section to meet its termination time. Whenever a section is considered for insertion in σ, it is scheduled to meet its own termination time. However, all of the sections in its dependency list must execute before it can execute, and therefore, must precede it in the schedule. The index values of the dependent sections may be changed with `Insert()` in line 17 of Algorithm 5.

The variable $CuTT$ keeps track of this information. It is initialized with the termination time of section S_i, which is tentatively added to the schedule (line 7). Thereafter, any section in $S_i.Dep$ with a later

termination time than $CuTT$ is required to meet $CuTT$ (lines 13; 16–17). If, however, a section has a tighter termination time than $CuTT$, then it is scheduled to meet that time (line 11), and $CuTT$ is advanced to that time since all sections left in $T_i.Dep$ must complete by then (lines 16–17).

When S_i (or any section $S_j \in S_i.Dep$) is inserted in σ, its handler S_i^h is immediately inserted to meet a termination time that is equal to S_i's termination time plus S_i^h's (relative) termination time (lines 6, 18). When a section in $S_i.Dep$ with a later termination time than $CuTT$ is advanced to meet $CuTT$, the section's handler is also correspondingly advanced (lines 14–15; 18).

Finally, if this insertion (of S_i, its handler, sections in $S_i.Dep$, and their handlers) produces a feasible schedule, then the sections are included in this schedule; otherwise, not (lines 19–20). If a rejected section S_i (due to schedule infeasibility in line 19) is not a head and belonged to the schedule σ_r constructed at the previous scheduling event, then the TPR protocol is notified regarding S_i's failure (lines 21–23).

Computational Complexity. With n sections, HUA's asymptotic cost is $O(n^2 \log n)$ (for brevity, we skip the analysis). Though this cost is higher than that of many traditional real-time scheduling algorithms, it is justified for dynamic applications with longer execution time magnitudes such as the NCW application in Section II. (Of course, this high cost cannot be justified for every application.)

G. Algorithm Properties

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

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

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

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

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

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

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

We now describe the NBI of HUA and other UA algorithms including DASA [12], LBESA [11], and AUA [15] (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 denotes HUA's schedule at time t. DASA's and LBESA's worst-case NBI is zero; AUA's is $+\infty$.

Proof: The time t that will result in the worst-case NBI for HUA is when $\sigma_t = H \neq \emptyset$. By NBI's definition, S_i has the highest PUD and is feasible. Thus, S_i will be included in the feasible schedule σ,

[4] If $\mathcal{A}$ does not consider a section's handler for feasibility (e.g., [11], [12]), the handler's execution time is regarded as zero.

resulting in the rejection of some handlers in H. Consequently, the algorithm will discard σ 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 σ_t complete (i.e., at handler completion times), as S_i and its handler may be infeasible with the remaining handlers in σ_t at each of those events. Since each handler in σ_t is scheduled to complete by $\max_{\forall S_j \in \sigma_t} \left(S_j.X + S_j^h.X\right)$, the earliest time that S_i becomes feasible is $t + \max_{\forall S_j \in \sigma_t} \left(S_j.X + S_j^h.X\right)$.

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

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

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

Proof: HUA's best-case NBI occurs when S_i arrives at t and the algorithm includes S_i and all handlers in H in the feasible schedule σ (thus HUA only rejects some *sections* in σ_t to construct σ). 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. ■

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

HUA produces optimum total utility for the following special case.

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

Proof: This is self-evident. For a thread without dependencies, the dependency list of each section S_i of the thread, $S_i.Dep$, only contains S_i. If there is sufficient processor time for meeting the termination times of all sections and their handlers on all nodes, then schedule σ_{tmp} will always be feasible in line 19 of Algorithm 5. Consequently, no section is rejected and the output schedule σ in line 26 of Algorithm 1 is termination time-ordered. The TUF termination time that we consider is analogous to the deadline in [6]. From [6], an EDF schedule is optimal (with respect to meeting all deadlines) during underloads. Thus, HUA's σ and system-wide schedule will yield the same total utility as EDF. ■

HUA also exhibits non-timeliness properties including freedom from local deadlocks, correctness (i.e., the resource requested by a section dispatched for execution by HUA is free), and mutual exclusion. These properties are self-evident from the algorithm description. For brevity, we omit their proofs.

V. The TPR Protocol

A. Overview

The TPR protocol is instantiated in a software component called the Thread Integrity Manager (TIM). Each node that hosts thread sections has a TIM component, which continually runs TPR's (phased) polling operation.

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

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

B. Thread Polling

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

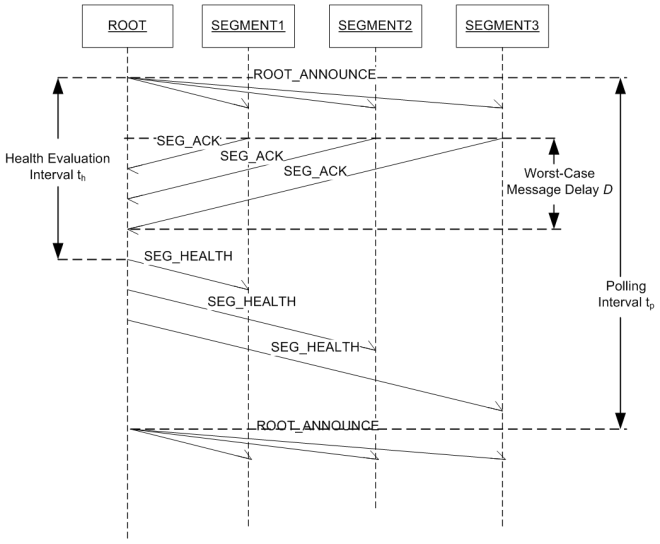

Fig. 3. TPR Operation — Healthy Thread

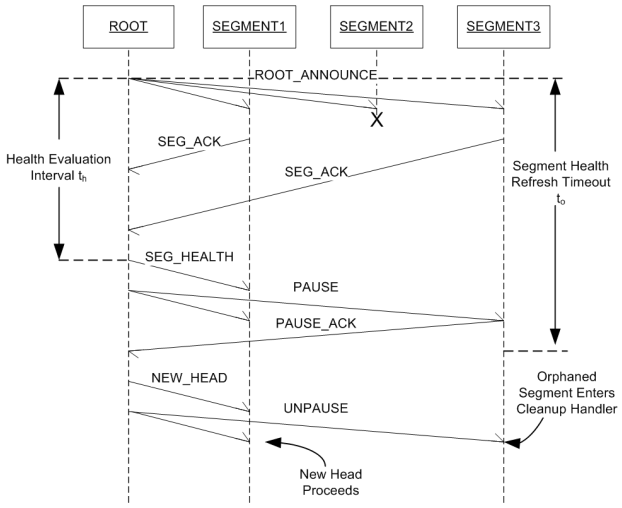

Fig. 4. TPR Operation — Unhealthy Thread Entering Recovery

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

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

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

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

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

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

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

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

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

Sections may thus evaluate their health at a constant interval, irrespective of the dynamics of the system.

C. Recovery

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

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

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

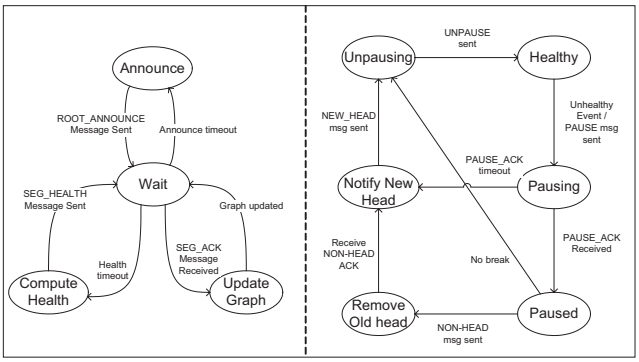

Fig. 5. High-level State Diagram – Root Section

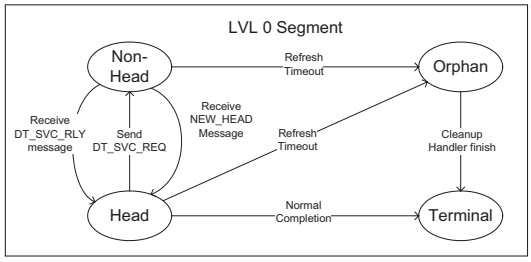

Fig. 6. High-level State Diagram – Section

Figure 5 illustrates the states experienced by a thread from the standpoint of its root section; Figure 6 illustrates the states from the standpoint of a section. In the first step, the recovery operation broadcasts

a PAUSE message and waits. The recovery thread continues waiting until it either receives a PAUSE_ACK message from the current head of the thread or a user-specified amount of time lapses without a PAUSE_ACK message being received.

In the second step, the recovery operation analyzes the thread's distributed call-graph and finds the contiguous thread section farthest from the root, which will be the new head. If the old head still exists after this step, the recovery thread must terminate the old head and wait for an acknowledgement that this action has been completed.

In the third step, the recovery thread sends a NEW_HEAD message to the node hosting the new head. In the fourth step, the recovery thread broadcasts an UNPAUSE message to all nodes within the system. The recovery operation then terminates, and the thread is considered healthy.

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

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

From here, the point of execution is to return to application code at the new head at the point of remote invocation. An error code is returned to indicate that a thread integrity failure has occurred, and it is the responsibility of the application programmer to decide how to proceed (e.g., resumption of thread execution).

D. Orphan Cleanup

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

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

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

VI. Implementation Experience

We implemented HUA and TPR in the Reference Implementation (RI) of the proposed DRTSJ [1]. The RI includes a user-space scheduling framework, called Metascheduler, for pluggable thread scheduling (similar to [23]) and mechanisms for implementing thread integrity protocols (e.g., TIM). The RI infrastructure runs atop a slightly modified version of Apogee's Aphelion Real-Time Java Virtual Machine (JVM) that is compliant with the Sun Real-Time Specification for Java (RTSJ). These modifications include modifications to the class library in support of the proposed DRTSJ specification as well as more aggressive changes to support experimental work on advanced pluggable and distributed scheduling policies in the RI. For example, the modified version of Apogees Aphelion JVM has hooks for notifying user-space schedulers of state changes in Java object monitors. This RTSJ platform runs atop the Debian Linux OS (kernel version 2.6.16-2-686) on a 800MHz, Pentium-III processor. Our experimental testbed consisted of a network with five such RI nodes.

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

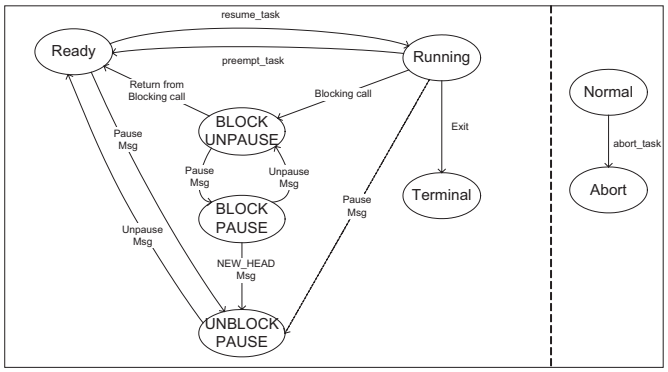

Fig. 7. Thread Scheduling States

In Figure 7, we present the various scheduling states supporting the distribution middleware. When a thread enters a PAUSE or BLOCK state, the scheduler is able to resolve resource contention and dependencies while respecting local mutual exclusion invariants. Furthermore, the PAUSE state is explicitly governed to allow coordinated control of all segments of a distributable thread.

Besides HUA, we implemented AUA [15]. This allows a comparison between HUA/TPR and AUA/TPR. Our test application was composed of one master node and four slave nodes. The master node was responsible for issuing commands to the slave nodes and logging events on a single timescale. The slave nodes were required to accept commands from the master node and were responsible for the execution, propagation, and maintenance of threads.

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

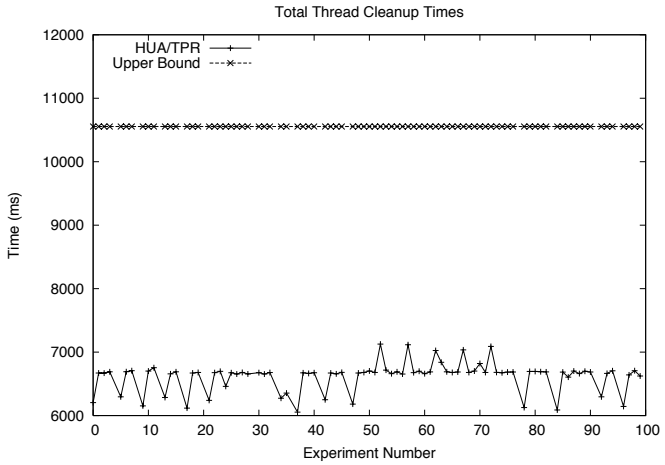

Fig. 8. Total Thread Cleanup Times

The Total Thread Cleanup Time is the time between the failure of a thread's node (causing a section failure) and the completion of the handlers of all the orphan sections of the thread. Figure 8 shows the measured cleanup time for HUA/TPR plotted against its cleanup upper bound time for the thread set used in our experiments. We observe that HUA/TPR satisfies its cleanup upper bound, validating Theorem 9.

In order for the Total Thread Cleanup Time to satisfy the HUA/TPR cleanup bound, TPR must detect a failure within a certain amount of time as determined by the protocol parameters (e.g., polling interval

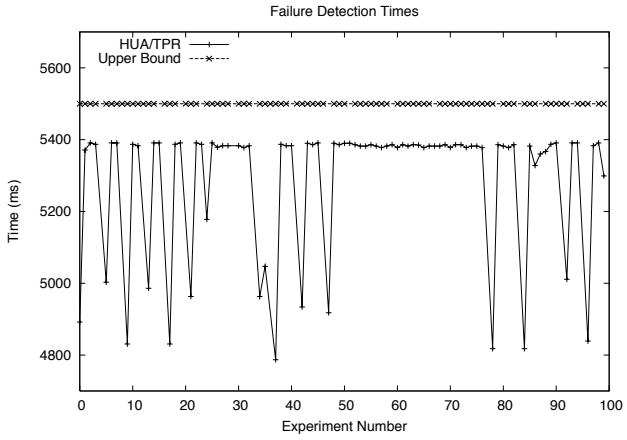

Fig. 9. Failure Detection Times

t_p). Figure 9 shows TPR's Failure Detection times as measured during failures in our test application and the upper bound on failure detection time as calculated using our experimental parameters. As the figure shows, TPR satisfies the upper bound on failure detection.

The variation observed in Figure 9 is actually less than the theoretic variation of t_p (1 second for these experiments) as described in Lemma 6. (This variation also occurs in Figure 10 for the same reason.)

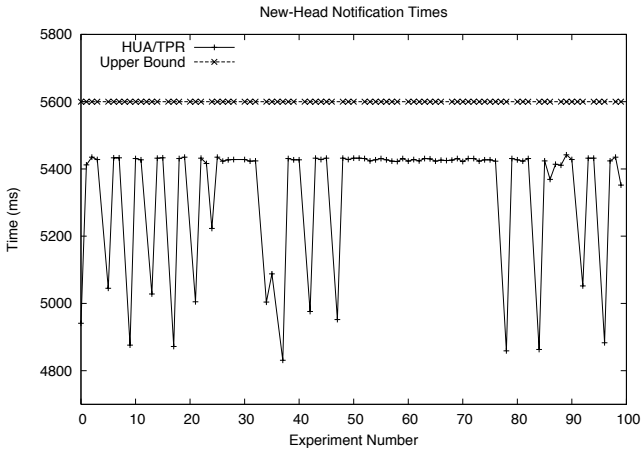

Fig. 10. New-Head Notification Times

For a thread to recover from a thread break, a new head must be established and orphans must be notified to clean themselves up. Therefore, the last time that must be bounded in order for HUA/TPR to achieve an upper limit on orphan cleanup is the time it takes for the protocol to determine and notify a thread of its new head. We measure this as the New-Head Notification Time. Figure 10 shows TPR's New-Head Notification Time and the notification time bound that TPR must satisfy in order to meet the Total Thread Cleanup bound. We observe that HUA/TPR satisfies the notification time bound.

Figure 11 shows the thread completion times of experiments 1) with failures and TPR, 2) without failures and without TPR, 3) without failures and without TPR, and 4) with failures and without TPR. By

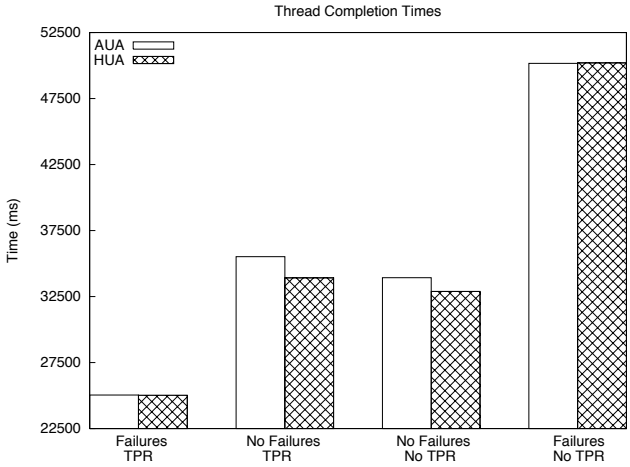

Fig. 11. Thread Completion Times

measuring the thread completion times under these scenarios, we measure the overhead of TPR in terms of the increase in thread completion times caused by the protocol operation.

Thread Completion Time is the difference between the time when a root section of a thread starts and the time when it completes. As orphan cleanup can occur in parallel with the continuation of a repaired thread, Thread Completion Time may ignore orphan cleanup times, making completion times of failed threads shorter than completion times of successful threads. This behavior is evident in Figure 11 as the experiments with failures and with TPR had the shortest completion times.

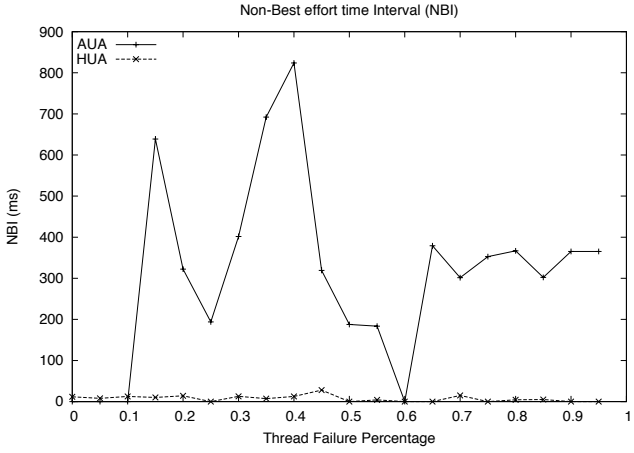

Fig. 12. Non-Best-effort time Interval (NBI)

One of the most interesting aspects of Figure 11 is the contrast between the experiments without failures. This contrast shows the overhead that TPR incurs when there are no failures present. Another interesting aspect of the figure is the large completion times for experiments with failures, but without TPR. The RI platform that we used for implementing HUA/TPR enforces a simple, tunable failure detection scheme in the absence of a thread integrity protocol. We purposely chose a large failure detection delay to convey the idea that the threads would never complete without any kind of failure detection and are subject to

longer than necessary completion times if the detection scheme is naive.

Figure 12 shows the NBI of AUA and HUA under an increasing number of thread failures. We observe that AUA has a higher NBI than HUA, validating Theorems 2 and 3. This difference in NBI is due to AUA's admission control policy [15]: AUA rejects threads arriving during overloads to respect the assurance made to previously admitted threads, irrespective of thread importance. HUA does not have this policy, and is generally free (limited by its NBI) to admit threads arriving during overloads that might have higher PUDs.

The variation of the NBI observed in Figure 12 is due to the way failures were experimentally created. The sets of failed threads were identical for all experiments at the same failure percentage. But they were not a strict subset of the sets of failed threads for experiments with higher failure percentages.

VII. Past and Related Efforts

The problem that this paper focuses is on: 1) how to maximize the total accrued utility of distributable threads, in the presence of node failures; 2) how to bound completion times of cleanup handlers that are triggered to clean up thread orphans due to node crashes; and most importantly, 3) how to bound the best effort property, defined through the non-best effort time interval (or NBI) metric.

The unique aspects of HUA's thread scheduling model include: 1) threads with TUF time constraints; 2) threads with arbitrary arrivals; and 3) threads with no worst-case execution time knowledge (which can cause overloads

The majority extant results in the distributed real-time literature do not address this problem space. For example, most of them focus on deadline time constraints and optimizing deadline-based optimality criteria, in particular on the hard real-time objective of meeting all deadlines. The most notable examples in this category include the work on distributed real-time schedulability analysis for fixed priority systems [24] and that for EDF-scheduled systems [25]–[29]. Most of these results are based on the concept of holistic schedulability analysis, first presented in [30]. Tindell and Clark's results in [30] have been the basis for many of the distributed real-time schedulability efforts in the literature.

Tindell and Clark's holistic analysis [30], the semantic forerunner of many current schedulability analysis techniques for distributed real-time systems, was designed primarily to show-case the flexibility of fixed priority scheduling. For such systems, it allows real-time practitioners to derive upper bounds on response times by considering the entire system in an iterative analysis that incrementally converges towards a stable solution (precedence constraints are described using jitter). This iterative process, necessary because the response times of tasks in a distributed system are interdependent, is guaranteed to converge since it is monotonic in its parameters. One of its disadvantages, though, is the fact that it is a relatively pessimistic analysis technique that over-estimates response times. In addition, Tindell's holistic analysis was designed for fixed priority systems. In [25], Spuri shows how this technique can be extended to systems scheduled using the EDF policy.

There have been many attempts to reduce the pessimism of holistic analysis (e.g. [26]–[29]). Most of these attempt to alleviate the pessimism by including additional constraints on the task release times in order to reduce the worst case number of tasks that can execute concurrently (thus reducing *computed* system load and *computed* response times). For example, in [26] the authors develop an extension to the fixed priority holistic approach that allows for dynamic offsets. Using this approach, the authors show how increased utilization can be achieve by having a more realistic picture of when tasks will execute (the offsets enforce the fact that certain tasks cannot execute concurrently in reality). The same authors then extend this analysis to deal with EDF scheduled systems in [28] and to systems scheduled with EDF within fixed priorities in [27]. This approach was further refined in [29] where the authors show how the analysis can be further tightened using a slightly more computationally complex analysis technique. There are many more results in a similar vein, e.g. [31], which attempt to add some feature or make a refinement to this general approach.

Not all past works have been of the holistic nature; for example, Sun's work [24] is a classic example of an integrated distributed real-time system that did not use the holistic approach. Rather, the author decomposes the problem of end-to-end scheduling into three separate sub-problems. First, deriving priorities for tasks, then deriving appropriate release times (the author uses the term execution synchronization) to ensure that precedence constraints are met, and finally, performing schedulability analysis on the tasks on each node independently using the derived release times. The response times of sub-tasks belonging to an end-to-end computation abstraction are then summed to give an upper bound on the end-to-end response time. Note that this is not a holistic technique since each processor is considered on its own (after the release times have been derived to preserve precedence constraints). It should also be noted that Sun's approach involves "static release" of sub-tasks (i.e., the release times are computed offline and are enforced during run-time). This contrasts with the "dynamic release" approach of holistic analysis where release times are not set offline, but tasks are released upon the receipt of an invocation call at run-time and uncertainty about release times, as mentioned before, is described using jitter. This approach is generally considered less flexible than the holistic approach since its static release nature makes it rather restrictive for many modern systems where dynamics are a primary characteristic. In addition, the statically derived release times are also subject to pessimism.

The fundamental premise of these works is that task arrival and execution time behaviors are bounded, time constraints are deadlines, no overloads or node failures are presumed to occur, and the objective is to meet all deadlines. Thus, these efforts focus on deriving schedulability analysis conditions i.e., analytical conditions under which all deadlines can be satisfied. In contrast, our work's premise violate all those premises: task arrival and execution time behaviors are uncertain, time constraints are TUFs, overloads and node failures are common, the goal is to maximize total utility, besides bounding the NBI. Due to this fundamental mismatch of our and these other works' premises, we believe that a direct comparison between the distributed schedulabaility analysis results and our results is inappropriate.

There also have been past distributed real-time efforts that have studied coping with node crash failures. Notable example efforts in this category include [32]–[34]. In [32], Aguilera et. al. describe the design of a fast failure detector for synchronous systems and show how it can be used to solve the consensus problem for real-time systems. The algorithm achieves the optimal bound for both message and time complexity for synchronous systems. In [33], Hermant and Le Lann present an asynchronous solution for the uniform consensus problem that does not make any assumptions on timing variables such as upper bounds on inter-process message delays. First, considering an asynchronous model, they prove the safety of the solution—i.e., all processes correctly agree on some value. Then, considering a partially synchronous model, they analytically determine computable functions for timing variables that can be instantiated with known workload and failure hypothesis. In [34], Hermant and Widder describe the Theta-model, where only the ratio, Theta, between the fastest and slowest message in transit is known. This increases the coverage of algorithms (designed under this model) as fewer assumptions are made about the underlying system. While Theta is sufficient for proving the correctness of such algorithms, an upper bound on communication delay is needed to establish timeliness properties. Thus, these efforts, though considering node failures, do not consider the properties like overloads and NBI that we focus on.

The distributed TUF scheduling results are, naturally, the ones that are most closely related to our work. Notable efforts in this category include the Alpha RTOS [2], Alpha's Thread Polling protocol [2], the Node Alive protocol [16], and adaptive versions of Node Alive [16]. Alpha's problem space overlaps with ours in terms of TUF time constraints, thread arrival and execution time behaviors, node failures, and overloads. However, none of Alpha's scheduling algorithms [11], [12] and thread integrity protocols [2], [16] provide time-bounded cleanup and bounded NBI. Our work builds upon our prior work in [15] that provides bounded thread cleanup. However, [15] suffers from unbounded loss of the NBI. In contrast, HUA/TPR provides bounded thread cleanup with bounded loss of the best-effort property, which is precisely the paper's contribution.

VIII. Conclusions and Future Work

We presented a distributable thread scheduling algorithm called HUA and a thread integrity protocol called TPR. We showed that HUA/TPR bounds (A) the completion times of handlers that are released for threads which fail during execution, and (B) the time interval for which a high importance thread arriving during overloads has to wait to be included in a feasible schedule. Our implementation experience using the RI of a proposed DRTSJ demonstrated that the proposed distributable thread model for the DRTSJ can be feasibly implemented, and also demonstrated the HUA/TPR algorithm/protocol's cost-effectiveness.

Property (A) is potentially unbounded for best-effort algorithms, and property (B) is potentially unbounded for admission control algorithms. By bounding (A) and (B), HUA/TPR places itself between the two models, allowing applications to exploit the tradeoff space.

Directions for future work include relaxing TPR's requirements for reliable communication with bounded latency, and considering ad-hoc network infrastructures.

Acknowledgements

This work was supported by the US Office of Naval Research under Grant N00014-00-1-0549 and by MITRE Corporation under Grant 52917. E. Douglas Jensen was supported by the MITRE Technology Program.

A preliminary version of this paper appeared as "On Best-Effort Real-Time Assurances for Recovering from Distributable Thread Failures in Distributed Real-Time Systems," *IEEE ISORC*, pages 344-353, May 2007.

References

[1] J. Anderson and E. D. Jensen, "The distributed real-time specification for java: Status report," in *JTRES*, 2006.
[2] J. D. Northcutt, *Mechanisms for Reliable Distributed Real-Time Operating Systems — The Alpha Kernel*, Academic Press, 1987.
[3] B. Ford and J. Lepreau, "Evolving mach 3.0 to a migrating thread model," in *USENIX Technical Conf.*, 1994, pp. 97–114.
[4] The Open Group, *MK7.3a Release Notes*, The Open Group Research Institute, Cambridge, Massachusetts, October 1998.
[5] OMG, "Real-time corba 2.0: Dynamic scheduling specification," Tech. Rep., Object Management Group, September 2001.
[6] W. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quaterly*, vol. 21, pp. 177–185, 1974.
[7] R. Clark, E. D. Jensen, et al., "An adaptive, distributed airborne tracking system," in *IEEE WPDRTS*, April 1999, pp. 353–362.
[8] D. P. Maynard, S. E. Shipman, et al., "An example real-time command, control, and battle management application for alpha," Tech. Rep. Archons Project Technical Report 88121, CMU CS Dept., December 1988.
[9] E. D. Jensen et al., "A time-driven scheduling model for real-time systems," in *IEEE RTSS*, Dec. 1985, pp. 112–122.
[10] B. Ravindran, E. D. Jensen, and P. Li, "On recent advances in time/utility function real-time scheduling and resource management," in *IEEE ISORC*, May 2005, pp. 55 – 60.
[11] C. D. Locke, *Best-Effort Decision Making for Real-Time Scheduling*, Ph.D. thesis, CMU, 1986, CMU-CS-86-134.
[12] R. K. Clark, *Scheduling Dependent Real-Time Activities*, Ph.D. thesis, CMU, 1990, CMU-CS-90-155.
[13] A. Bestavros and S. Nagy, "Admission control and overload management for real-time databases," in *Real-Time Database Systems: Issues and Applications*, chapter 12. Kluwer Academic Publishers, 1997.
[14] H. Streich, "Taskpair-scheduling: An approach for dynamic real-time systems," *Mini and Microcomputers*, vol. 17, no. 2, pp. 77–83, 1995.
[15] E. Curley, J. S. Anderson, B. Ravindran, and E. D. Jensen, "Recovering from distributable thread failures with assured timeliness in real-time distributed systems," in *IEEE SRDS*, 2006, pp. 267–276.
[16] J. Goldberg, I. Greenberg, et al., "Adaptive fault-resistant systems (chapter 5: Adpative distributed thread integrity)," Tech. Rep. csl-95-02, SRI International, January 1995, `http://www.csl.sri.com/papers/sri-csl-95-02/`.
[17] Jeffrey R. Cares, *Distributed Networked Operations: The Foundations of Network Centric Warfare*, iUniverse, Inc., 2006.
[18] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.
[19] P. Li, *Utility Accrual Real-Time Scheduling: Models and Algorithms*, Ph.D. thesis, Virginia Tech, 2004.
[20] B. Ravindran, J. Anderson, and E. D. Jensen, "On distributed real-time scheduling in networked embedded systems in the presence of crash failures," in *IFIP SEUS Workshop, IEEE ISORC*, May 2007.
[21] D. L. Mills, "Improved algorithms for synchronizing computer network clocks," *IEEE/ACM TON*, vol. 3, pp. 245–254, June 1995.
[22] B. Kao and H. Garcia-Molina, "Deadline assignment in a distributed soft real-time system," *IEEE TPDS*, vol. 8, no. 12, pp. 1268–1274, Dec. 1997.
[23] P. Li, B. Ravindran, et al., "A formally verified application-level framework for real-time scheduling on posix real-time operating systems," *IEEE Trans. Software Engineering*, vol. 30, no. 9, pp. 613 – 629, Sept. 2004.
[24] J. Sun, *Fixed Priority Scheduling of End-to-End Periodic Tasks*, Ph.D. thesis, CS Dept., Univ. of Illinois at Urbana-Champaign, 1997.
[25] M. Spuri, "Holistic analysis of deadline scheduled real-time distributed systems," Tech. Rep. RR-2873, INRIA, France, 1996.

[26] J. C. Palencia and M. González Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, 1998, p. 26.

[27] M. González Harbour and J. C. Palencia, "Response time analysis for tasks scheduled under edf within fixed priorities," in *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, 2003, p. 200.

[28] J. C. Palencia and M. González Harbour, "Offset-based response time analysis of distributed systems scheduled under edf," in *15th Euromicro Conference on Real-Time Systems (ECRTS)*, 2003, pp. 3– 12.

[29] Rodolfo Pellizzoni and Giuseppe Lipari, "Improved schedulability analysis of real-time transactions with earliest deadline scheduling," in *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, 2005, pp. 66–75.

[30] Ken Tindell and John Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocess. Microprogram.*, vol. 40, no. 2-3, pp. 117–134, 1994.

[31] Romulo Silva de Oliveira and Joni da Silva Fraga, "Fixed priority scheduling of tasks with arbitrary precedence constraints in distributed hard real-time systems," *Journal of Systems Architecture*, vol. 49, no. 11, pp. 991–1004, 2000.

[32] Marcos Kawazoe Aguilera, Gérard Le Lann, and Sam Toueg, "On the impact of fast failure detectors on real-time fault-tolerant systems," in *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*. 2002, pp. 354–370, Springer-Verlag.

[33] Jean-François Hermant and Gérard Le Lann, "Fast asynchronous uniform consensus in real-time distributed systems," *IEEE Trans. Comput.*, vol. 51, no. 8, pp. 931–944, 2002.

[34] Jean-François Hermant and Josef Widder, "Implementing reliable distributed real-time systems with the *theta*-model," in *OPODIS*, 2005, pp. 334–350.