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

Edward A. Curley IV

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Peter M. Athanas
Amitabh Mishra
E. Douglas Jensen

February 2, 2007
Blacksburg, Virginia

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

Edward A. Curley IV

(ABSTRACT)

This thesis considers the problem of recovering from failures of distributable threads with assured timeliness. When a node hosting a portion of a distributable thread fails, it causes *orphans*—i.e., thread segments that are disconnected from the thread's root. A termination model is considered for recovering from such failures. In this model the orphans must be detected and cleaned up, and failure-exception notification must be delivered to the farthest, contiguous surviving thread segment for resuming thread execution. Two real-time scheduling algorithms (AUA and HUA) and three distributable thread integrity protocols (TPR, D-TPR and W-TPR) are presented. We show that AUA combined with any of the protocols presented bounds the orphan cleanup and recovery time, thereby bounding thread starvation durations and maximizing the total thread accrued timeliness utility. The algorithms and the protocols are implemented in a real-time middleware that supports distributable threads. The experimental studies with the implementation validate the algorithm/protocols' time-bounded recovery property and confirm their effectiveness.

Acknowledgments

I would like to thank Dr. Binoy Ravindran for all his guidance and assistance while working on this project. I would also like to thank Jonathan Anderson for the use of his distributable thread framework implementation and all his technical expertise in setting up the testbed for the experiments outlined in this report.Without their support, this would not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many distributed systems are most naturally reasoned about in terms of asynchronous concurrent sequential flows of execution within and among objects. The *distributable thread* programming model supported in OMG's recent Real-Time CORBA 1.2 standard (abbreviated here as RTC2) [28] and Sun's upcoming Distributed Real-Time Specification for Java (DRTSJ) standard [17] directly provides that as a first-class abstraction. Distributable threads first appeared in the Alpha OS [26, 16] and later in The OSF Research Institute's MK7.3 OS [30].

A distributable thread is a single thread of execution with a globally unique identifier that transparently extends and retracts through local and remote objects. Thus, a distributable thread is an end-to-end control flow abstraction, with a logically distinct locus of control flow which moves within and among objects and nodes, often through the use of Remote Procedure Calls (RPCs) and Remote Method Invocations (RMIs). For the remainder of this thesis, distributable threads will be referred to simply as *threads*, except as necessary for clarity.

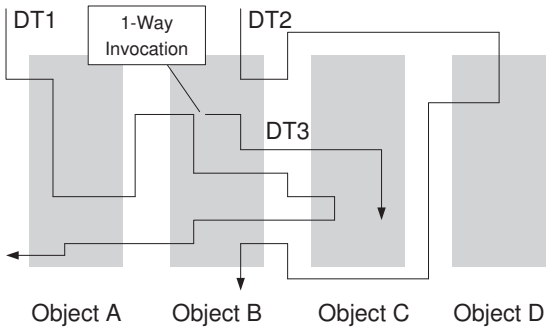

Figure 1.1: Distributable Threads

A thread carries its execution context as it transits node boundaries, including its scheduling parameters (e.g., time constraints, execution time), identity, and security credentials. Hence, threads require that Real-Time CORBA's *Client Propagated* model be used, not the *Server Declared* model. The propagated thread context is used by node schedulers for resolving all node-local resource contention among threads for node's physical (e.g., CPU, I/O) and logical (e.g., locks) resources, and for scheduling threads to optimize system-wide timeliness. Thus, threads constitute the programming abstraction for concurrency and scheduling. Figure 1.1 cited from [28] shows the execution of threads.

The Real-Time CORBA specification envisions four distributed scheduling "cases", summarized in Table 1. This thesis explicitly supports distributed scheduling schemes corresponding to Case 1 (in the case of local use of the AUA and HUA protocols) and Case 2 (for distributed threads). While the Real-Time CORBA specification does not address thread integrity concerns in any detail, it might be argued that the thread integrity protocols discussed in this thesis amount to forms of distributed resource management (specifically in the presence of partial failures) properly classified under Cases 3 or 4.

Experience with Operating Systems and middleware that directly provide the thread abstraction show that providing the distributable thread programming abstraction can reduce application development and maintenance costs. This is in contrast with situations where a

Table 1.1: Real-Time CORBA Distributed Scheduling Cases [28, Section 3.8]

Case 1	Scheduling decisions take place independently on each node.
Case 2	Scheduling decisions take place independently on each node, subject to time constraints which propagate between nodes with application activities.
Case 3	Scheduling decisions are made by a distributed scheduling algorithm with instances on each node. Local scheduler instances collaborate to achieve or approximate global optimality.
Case 4	Scheduling is hierarchical, with higher-level schedulers above case 1 or 2 instances which seek to improve resource allocation decisions with some global knowledge.

similar trans-node, control flow abstraction must be implemented from scratch by the programmer using lower level abstractions such as RPCs/RMIs, OS threads, locks, etc., and end-to-end properties of the flows have to be maintained such as assuring satisfaction of end-to-end time constraints, ensuring the integrity of the sequential operation flow through distributed node and link failure detection and recovery, and ensuring system safety through distributed deadlock detection and recovery.

This thesis focuses on real-time distributed systems that operate in environments with dynamically uncertain properties. These uncertainties include both transient and sustained resource overloads (due to context-dependent activity execution times), arbitrary arrival patterns for application activities, and arbitrary node/link failures.[1] Nevertheless, such systems require the strongest possible assurances on activity timeliness behavior that are feasible under the circumstances. Another important distinguishing feature of most of these systems is their relatively long activity execution time magnitudes, compared to those of conventional real-time subsystems—e.g., in the order of milliseconds to minutes. Some examples of such dynamic systems that motivate our work (from the defense domain) include phased array

[1]We consider only a crash (fail/stop) failure model in the case of node failures.

radars [12]), surveillance aircraft [13, 11, 3]), and network-centric warfare systems [10, 2]). These dynamic systems have traditionally been designed as hard, real-time systems, requiring worst-case load and failure models. Designers and users of these systems have found that, due to the long lifetimes of these systems, the increasingly dynamic execution environment, and the flexible way they are employed in real-world situations, the deterministic worst-case analysis performed at design and implementation time enforces unacceptable bounds on the use of the system.

1.1 Contributions: Time-Bounded Thread Maintenance and Recovery

When nodes transited by distributable threads fail, this may cause threads that span the nodes to break by dividing them into several pieces. Sections of a thread that are disconnected from the thread's node of origin (called the thread's *root-node*), are called *orphans*. In order to provide the abstraction of a continuous reliable thread, orphan sections of the thread must be detected and aborted, resources held by them must be released and rolled back to safe states, and a failure exception must be delivered to the farthest execution point of the surviving portion of the thread—i.e., the farthest contiguous thread section from the thread's root. Then, the computation may be allowed to continue. This thesis focuses on such a termination exception handling model, as that is consistent with most concurrent programming paradigms (e.g., Ada, Java).[2]

As real-time threads are subject to time constraints, orphan cleanup and removal must be done in a timely manner. For example, cleanup and removal of orphans of a failed

[2]Under a continuation model, the orphan sections may be allowed to continue execution transparently, after cleanup has completed. A discussion of termination versus continuation (or "resumption") models is beyond the scope of this thesis. See [9] for a complete treatment of this topic.

low urgency/low importance thread must cause minimal interference to high urgency/high importance threads. On the other hand, if orphans of a failed low urgency/low importance thread hold resources which are blocking high urgency/low importance threads, then the cleanup activity must have execution eligibility that reflects the urgency/importance of the blocked threads. Furthermore, once a failure occurs, the time interval between detection of the thread failure and the recovery of the thread must be bounded. In this instance, we assume recovery to be the delivery of failure notification (e.g., exception) to the farthest, contiguous surviving thread section. If this time interval is unbounded, broken threads cause starvation—e.g., threads blocked on resources held by orphans may never be unblocked if those orphans are never cleaned up.

In the model considered herein, it is explicitly assumed that thread overruns are at best wasteful, and at worst degrade system safety. Thus, the scheduling algorithm must ensure that overruns do not occur. Similarly, this model assumes that the application-specified abort code (if any) is unexceptionally the correct response to overrun or failure conditions. If the application wishes to schedule request new, complex, and less restricted handler, it is free to spawn such an activity as a separate thread.[3]

We present two real-time scheduling algorithms called *Abort-Assured Utility Accrual Scheduling Algorithm* and *Handler-Assured Utility Accrual Scheduling Algorithm* (AUA and HUA, respectively) and three thread integrity protocols called *Thread Polling with Bounded Recovery* (or TPR), *Decentralized Thread Polling with Bounded Recovery* (or D-TPR), and *Wireless Thread Polling with Bounded Recovery* (or W-TPR). The algorithms and protocols are designed to achieve the above objectives in an appropriate manner for the RTC2/DRTSJ distributable threads programming model. End-to-end time constraints are specified for threads using Jensen's Time/Utility Function (or TUF) [15] time constraint model, which generalizes

[3]These assumptions are not always the correct ones for all systems.

the classical deadline time constraint and decouples thread urgency from thread importance. This decoupling facilitates thread scheduling that favors more important threads over less important ones, irrespective of their urgency. This is particularly important during overloads when all threads cannot be completed. Threads may be created at arbitrary times, and may span nodes that are subject to arbitrary crash failures.

AUA and HUA are shown to achieve optimal total accrued utility during the special case of underloads and no failures, and maximize the total utility to the extent possible during overloads and failures. It is further established that AUA, in conjunction with TPR, D-TPR, or W-TPR, bounds cleanup and recovery times, thereby bounding thread starvation durations. The experimental measurements from the implementations of AUA, TPR, D-TPR, and W-TPR in RTC2-like real-time middlewares validate the algorithm/protocol properties and confirm their effectiveness.

Thread integrity protocols (Thread Polling [6, 27], Node Alive [14], and their adaptive versions [14]) and real-time scheduling algorithms (DASA [4], D_OVER [20], EDF [23], LBESA [24], etc.) have been developed in the past . However, to the best of our knowledge, no thread integrity solution—i.e., a stand-alone protocol or one that is coupled with a scheduling algorithm — exists which provides end-to-end time-bounded cleanup and recovery. It is precisely this open research problem which is addressed by this thesis. Thus, our central contribution is time-bounded cleanup and recovery for LAN and MANET networks provided by the AUA algorithm when used in conjunction with the TPR, D-TPR, and W-TPR protocols.

The rest of the thesis is organized as follows: In Chapter 2, the models of the work are discussed and the algorithm/protocol objectives are stated. Chapter 3 presents the AUA algorithm, Chapter 4 presents the HUA algorithm, Chapter 5 presents the TPR protocol, Chapter 6 presents the D-TPR protocol, and Chapter 7 presents the W-TPR protocol. In

Chapter 8, AUA, HUA, and the thread integrity protocols are evaluated . The thesis is concluded and future work is identified in Chapter 9.

Chapter 2

Models and Objectives

2.1 Distributable Thread Abstraction

Distributable threads execute in local and remote objects by location-independent invocations and returns. A thread begins its execution by invoking an object operation. The object and the operation are specified when the thread is created. The portion of a thread executing an object operation is called a *thread segment* and a maximal set of contiguous segments on a single node is called a *thread section*. Thus, a thread can be viewed as being composed of a concatenation of thread sections across one or several nodes.

To avoid confusion, the notation $S_{i,j}$ will be used to denote the *section* i of thread j and $s_{i,j}$ will be used to denote the *segment* i of thread j. Thus, a thread can be viewed as a concatenation of thread *segments* or as a concatenation of thread *sections*.

$$DT_j = \{S_{i,j} : 1 \leq i \leq N\}$$

$$DT_j = \{s_{i,j} : 1 \leq i \leq n\}$$

In the notation above, n is the number of *segments* in the thread, N is the number of *sections* in the thread, and j is the distributed thread identifier.

A thread's initial segment is called its *root* (s_0) and its most recent segment is called its *head* (s_n). In healthy threads the head of a thread is the only segment that is active. The first segment in a section results either from an invocation from another node or the creation of distributed thread. The last segment in a section is either the execution point of the thread or is a section that has performed a remote invocation and is waiting on a return. Figure 2.1 illustrates threads, sections, and segments.

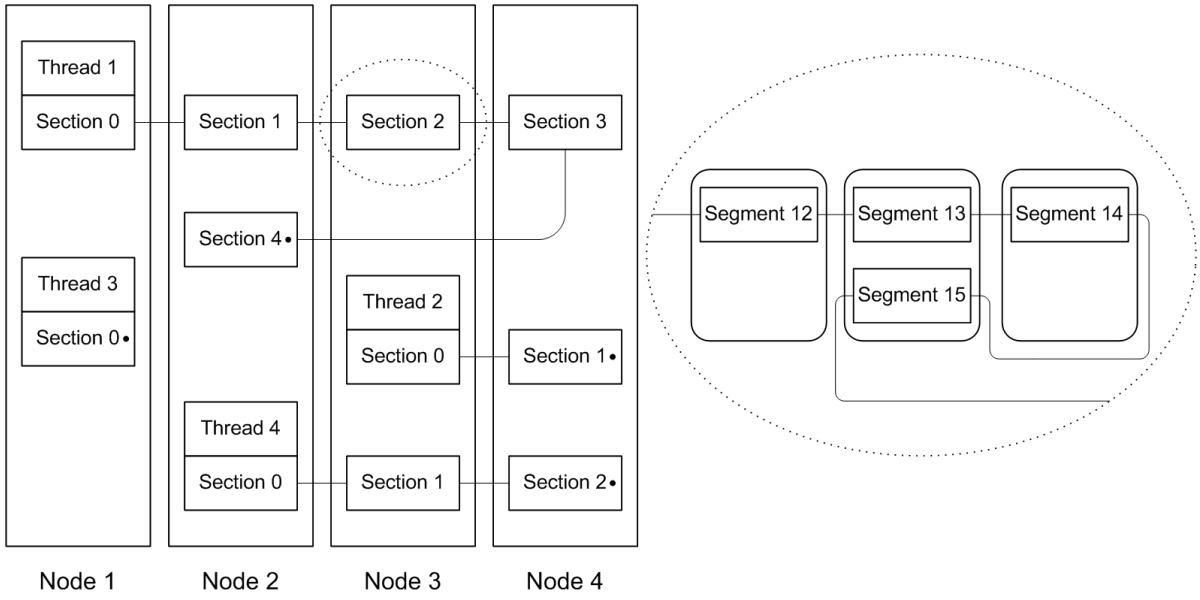

Figure 2.1: Distributable Threads, Segments, and Sections. Derived from [14]

Figure 2.1 shows four distributed threads where each thread starts with the section labeled "Section 0", its root-section. Sections within the figure that contain a "·" are sections which contain the thread's head. This figure shows how a distributed thread propagates over several nodes.

In the context of a distributable thread, each remote invocation is composed of two segments: (s_i) and (s_{i+1}). The predecessor segment makes a remote invocation on a remote object

and the segment that is created to execute that remote invocation on the remote object is called the successor. In most instances, the predecessor will be located on a different node from the successor. In this regard, nodes participating in a remote invocation have a predecessor/successor relationship with respect to the distributable thread making the remote invocation.

In Figure 2.1, the predecessor node of Node 2 (with respect to Thread 1) is Node 1. Likewise, the predecessor section of Section 3 of Thread 1 is Section 2 of Thread 1. Conversely, the successor section of Section 3 of Thread 1 is Section 4. In the current model, each section can only have at most one predecessor and at most one successor. The head section has no successor, the root segment has no predecessor, and all other segments have both.

Threads may be created at any node at any time. Upon arrival at a node, threads are assumed to present to the scheduler execution time estimates of normal code and cleanup code (or exception handler code) for segments of the thread at that node. While execution time estimates of normal code may be violated at run-time (e.g., due to context dependence), estimates of cleanup code may not as they are assumed to be non-context-dependent.

When a single failure occurs, the thread is temporarily or permanently split into two smaller, connected sets of sections. A section (S_i) is connected to another section (S_{i+1}) if both sections can communicate with each other (i.e., S_{i+1} can send and receive messages from S_i and S_i can send and receive messages from S_{i+1}). Disconnected sections are those that cannot communicate successfully both upstream and downstream (e.g., S_i is unable to receive messages from S_{i+1}).

For the purposes of this thesis, these sets of connected sections will be referred to as *pieces*. A piece of a distributed thread is a maximal set of contiguous, connected sections in which the first section in the set is disconnected from or has no predecessor and the last section in

the set is disconnected from or has no successor. Therefore, a root section will always be the first segment of a piece and the head section will always be the last segment of a piece.

A piece is composed of three components, a root-ward section (S_l), a head-ward section (S_r), and zero or more middle sections (S_m). S_l must be connected to some section $S_x : x = min(r, l+1)$ and disconnected from S_{l-1}. Likewise, S_r must be connected to some section $S_y : y = max(l, r-1)$ and disconnected from S_{r+1}. The set of middle sections may be denoted $S_m = \{S_i : l < i < r, S_i \text{ can communicate fully with both } S_{i+1} \text{ and } S_{i-1}\}$. A piece of a thread can be defined as $P_{l,r} = S_l \bigcup S_r \bigcup S_m$. As it is assumed that every section is always connected to itself, it is possible to have a piece composed of a single section where $S_l \equiv S_r$.

The set of distributed threads within a system shall be denoted $\mathbf{T} = \{T_k : 1 \leq k \leq n\}$.

2.2 Timeliness Model

We specify the time constraint of each thread using a TUF. A TUF specifies the utility of completing a thread as a function of its completion time. Figure 2.2 shows downward "step" shaped TUFs.

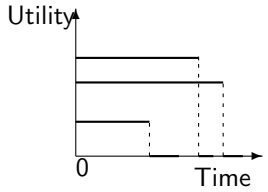

Figure 2.2: Example Step TUFs

A TUF decouples importance and urgency of a thread—i.e., urgency is measured as a deadline on the X-axis, and importance is denoted by utility on the Y-axis. This decoupling (see Figure 2.2) is a key property of TUFs, as a thread's urgency is typically orthogonal to its relative importance—e.g., the most urgent thread can be the least important, and vice

versa; the most urgent can be the most important, and vice versa. The more commonly used priority/deadline approaches fail to make this distinction.

A thread T_i's TUF is denoted as $U_i(t)$. A classical deadline is unit-valued—i.e., $U_i(t) = \{0, 1\}$, since importance is not considered. Downward step TUFs (Fig. 2.2) are a generalization of classical deadlines where $U_i(t) = \{0, \{n\}\}$. We focus on downward step functions, and denote the maximum, constant utility of a TUF $U_i()$, simply as U_i.

Each TUF has an initial time I_i, which is the earliest time for which the TUF is defined, and a critical time X_i, which, for a downward step TUF, is its discontinuity point. We assume that $U_i(t) > 0, \forall t \in [I_i, X_i]$ and $U_i(t) = 0, \forall t \notin [I_i, X_i], \forall i$.

In the policy we present, if a thread's critical time is reached and its execution has not been completed, a failure-exception is raised, and exception handlers are released for cleaning up all partially executed thread sections (by releasing system resources). The handlers' time constraints are also specified using TUFs.

When a thread T_i arrives after the failure of a thread T_j but before the completion of T_j^h, HUA *may* exclude T_i from a schedule until T_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 1. *Consider a scheduling algorithm $\mathcal{A}$. Let a thread T_i arrive at a time t with the following properties: (a) T_i and its handler together with all threads in $\mathcal{A}$'s schedule at time t are not feasible at t, but T_i and its handler are feasible just by themselves;*[1] *(b) One or more handlers (which were released due to thread failures before t) have not completed their execution at t; and (c) T_i has the highest PUD among all threads 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 T_i will have to*

[1] If $\mathcal{A}$ does not consider a thread's handler for feasibility (e.g., [24, 5]), then the handler's execution time is regarded as zero.

wait after t, before it is included in $\mathcal{A}$'s feasible schedule. Thus, T_i is assumed to be feasible together with its handler at $t + NBI_{\mathcal{A}}$.

As an example, we will describe the NBI of DASA and LBESA. Both DASA and LBESA will examine T_i at t, since a task arrival is always a scheduling event for them. Further, since T_i has the highest PUD and is feasible, they will include T_i in their feasible schedules at t (before including any other tasks), yielding a zero worst-case NBI. For the same reason, the best-case NBI for DASA and LBESA are both zero.

The NBI of AUA and HUA will be described in further detail in Chapters 3 and 4, respectively.

2.3 System Model

We consider a system model wherein a set of processing components generically referred to as *nodes* are interconnected via a network. Each node executes thread sections. The order of executing sections on a node is determined by the scheduler residing at the node. We consider RTC2's *Case 2* approach [28] for thread scheduling. According to this approach, node schedulers use the propagated thread scheduling parameters and independently schedule thread sections on respective nodes to optimize the system-wide timeliness optimality criterion. Thus, scheduling decisions made by a node scheduler are independent of other node schedulers. Though this results in approximate, global, system-wide timeliness, RTC2 supports the approach due to its simplicity and capability for coherent end-to-end scheduling. The approach's effectiveness is illustrated in Alpha OS [16] and Tempus middleware [21].[2]

[2]RTC2 also describes Cases 1, 3, and 4, which describe non real-time, global and multilevel distributed scheduling, respectively [28]. However, RTC2 does not support Cases 3 and 4.

2.3.1 Networks

Two network models are considered in this thesis. The first is a single hop network model (e.g., a LAN), where nodes are interconnected through a hub or a switch. This model presumes the existence of a reliable message transport with worst case message delivery latency D for administrative level communications. All other, non-administrative communications will contend for the network links. Such contentions must be resolved and packets must be scheduled on network links using a packet scheduling algorithm. We do not consider any particular algorithm for scheduling packets; the scheduling algorithms and thread integrity protocols presented are independent of any such algorithm.

We denote the set of nodes as $P_i \in P, i \in [1, m]$. We assume that all node clocks are synchronized using a protocol such as [25]. We consider an arbitrary, crash failure model for the nodes—i.e., any node can fail at any time and when it does so, it simply halts.

The second network model consists of a set of nodes, denoted $N = \{n_1, n_2, n_3, ...\}$, communicating through bidirectional wireless connections. A basic unicast routing protocol is assumed to be available for packet transmission. Examples of unicast routing protocols include DSR [18] and OLSR [7]. MAC-layer packet scheduling is assumed to be done by a CSMA/CA-like protocol (e.g., IEEE 802.11). Node clocks are synchronized using an algorithm such as the one described in [29]. Nodes may dynamically join/leave the network or may fail by crashing, links may fail transiently or permanently, and messages may be lost; all arbitrarily.

2.4 Failure Models

This thesis will limit its consideration of failures to catastrophic (fail-stop) node failures and communication failures.

A node will be considered to have failed when it is assumed to be permanently disconnected from the rest of the network. The node becomes disconnected when it can neither send nor receive messages from any other node. The term "permanently" implies that once the node has been partitioned from the rest of the network, it will never again be able to communicate with any other node.

A communication failure occurs when any message does not reach its desired destination. Failure of a message to reach its destination can occur for a number of reasons: a physical break in the communication medium, communication buffer overflow, parity error, etc. Depending on how many consecutive errors occur, the set of errors may be labeled *persistent* or *transient*. A relatively low number of continuous communication errors would be considered a transient error as it is assumed to be temporary. As the number of sequential errors increases, the error may be considered persistent and therefore permanent. The criteria by which an error is deemed transient or permanent is application specific. In terms of real-time applications, this criteria can be associated with the application's time constraints.

This thesis avoids the assumption that communication errors are bidirectional. If a communication error occurs between node A and node B, communications from node B to node A may not fail. This is of particular importance in wireless systems where message delivery failures are common and may not be considered "errors".

2.5 Thread Recovery and Cleanup

We consider thread recovery to be the process of notifying the farthest contiguous surviving thread segment that it may continue execution and we define orphan cleanup as the process of aborting and executing handlers for all orphans. The total "orphan cleanup and recovery time" is the maximum time required by the protocol to perform both recovery and orphan cleanup after a failure is detected.

Ordered cleanup within the context of a non-distributable thread, such as is provided by exception processing, is the Last-In-First-Out (LIFO) cleanup of the procedure calls on a thread's stack. In the case of distributable threads, the "stack" is distributed across the system, including both local and remote invocations. Therefore, it stands to reason that ordered cleanup for a distributable thread would be the LIFO cleanup of remote invocations on a distributable thread's stack. However, as distributable threads are subject to errors that may make portions of the DT's stack inaccessible (communication/node failures), a more appropriate definition for ordered cleanup of a distributed thread is required.

We define ordered cleanup as the LIFO cleanup of contiguous portions of the distributed thread's stack. Another way of describing this is that each thread piece (as described in section 2.1) will be cleaned up in a LIFO-ordered fashion, starting with the farthest downstream segment and working upstream (towards the root). This strays slightly from the non-distributed definition of ordered cleanup as no guarantee is provided regarding ordering between pieces. However, given the possible ways a thread can be partitioned, it is an acceptable divergence.

Ordered cleanup helps prevent deadlock when using remote resources. It also strengthens the distributable thread abstraction by more closely mimicking the cleanup process of single-node threads. Providing ordered cleanup also gives the application programmer a consistent

and familiar failure model to work with so that they might more easily deal with errors unique to distributable threads.

2.6 Objectives

Our objectives are to 1) maximize the total thread accrued utility, 2) bound the orphan cleanup and recovery time, and 3) provide best-effort ordered-cleanup of orphans. Note that maximizing the total utility subsumes meeting all TUF critical times as a special case. When all critical times are met (which is only possible during underload), the total utility is the optimum possible. During overload, we seek to maximize the total utility to the extent possible, since not all critical times can be met.

Chapter 3

The AUA Algorithm

3.1 Rationale

In order to attain bounded recovery time for distributable threads, it is necessary to have a scheduling algorithm which guarantees a bound on the time required by each *orphaned* thread section to detect and conduct cleanup operations. Without this guarantee, it would be possible for a broken thread to leave the system in an unsafe state. In particular, it would be possible for a single thread to have multiple, uncoordinated points of execution for an unbounded amount of time. In order to facilitate this guarantee, we have developed the AUA scheduling algorithm, described in section 3.2.

The AUA scheduling algorithm is a hybrid approach, seeking to maximize the total (summed) utility for all tasks in the system, subject to the guarantee that execution of those blocks of code designated as cleanup handlers will always complete by their TUF critical time (or deadline). As such, traditional "hard real-time" analysis techniques may be applied to this (typically small) subset of the application code. In particular, these guarantees are exploited

by the thread integrity protocols presented in the later chapters.

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

AUA traces its lineage to the *Dependent Activity Scheduling Algorithm* (DASA) introduced by Clark [4], and is equivalent to DASA if no abort handlers are introduced.[1]

3.2 Algorithm Overview

The AUA algorithm is presented in two parts, an event handler in Algorithm 1 and the core scheduling logic in Algorithm 2. When a thread section/handler pair is introduced to the system, the scheduler first checks to see if the handler's execution can be guaranteed within the time constraint. If not, the thread section/handler pair is rejected and no new schedule is created. If the new handler is accepted, its last-chance time (LCT) to commence execution is calculated by subtracting its Worst Case Excecution Time (WCET) from its deadline, allowing the scheduler to plan for the last moment at which the abort handler can be guaranteed to execute to completion.

In order to describe AUA, we introduce the notation given in Table 3.1 and the functions given in Table 3.2.

[1]This is not meant to imply that DASA is unable to handle aborts. While both DASA and AUA handle aborts, they do so using different approaches.

Table 3.1: Variables in AUA

T_r	current set of accepted, unscheduled threads
T_c	current handlers accepted in the system
T_{rnew}	new arriving thread
T_{cnew}	new arriving thread handler
T_i.DL	the thread's deadline for $T_i \in T_r$
T_i.UD	potential utility density of the thread / handler pair
T_i.Dep	the task that T_i is dependent on
T_i.LCT	the latest time at which the handler of T_i is guaranteed to complete before its deadline
σ	the current ordered schedule
σ_{copy}	temporary copy of current schedule. Used when updating schedule

Table 3.1 shows many of the variables used in the description of AUA. Thread variables are denoted by an uppercase T and schedule variables are denoted by a lowercase *sigma*. These conventions are also used in the operations described in Table 3.2.

Table 3.2: Operations used in AUA

inSchedule(T, σ)	return a boolean value indicating whether T is in schedule σ
headOf(σ)	return the first thread in schedule σ
sortByUD(σ)	return a new schedule sorted by non-increasing utility density (UD)
insertByEDF(T, σ)	insert thread T into ordered list σ by Earliest Deadline First; if there are already entries with the same deadline, T is inserted before them
remove(T, σ)	remove T from the ordered list σ if T is in σ
feasible(σ)	return a boolean value indicating schedule σ's feasibility. For σ to be feasible, the predicted completion time of each thread in σ must never exceed its deadline
setLCT(t)	set a timer to wake up the scheduler at time t

When the algorithm is invoked, it first computes a deadline ordered schedule of all the cleanup handlers accepted into the system. Then, depending on the scheduling event, it will do one of three things: attempt to add a handler to the system and create a new schedule, remove a handler from the system and create a new schedule, or schedule the handler with

Algorithm 1: AUA Algorithm

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

the earliest deadline for execution.

Scheduling events in AUA include: 1.) the arrival of a thread/handler pair, 2.) the completion of a thread, 3.) the completion of a handler, 4.) a resource request, 5.) a resource release, 6.) and the arrival of a handler's last-chance time (LCT). For clarity, Algorithm 1 presents only those events which directly impinge on AUA's performance. See [21] for a thorough description of resource request/grant event processing in the *Metascheduler*.

Once the event-specific actions have been executed, AUA proceeds to the utility accrual (UA) optimization step (shown in Algorithm 2).

As shown in Algorithm 2, thread sections are inserted into a schedule using a mechanism very similar to the DASA algorithm. In fact, this mechanism only differs from DASA when

handlers are associated with threads. Therefore, when no handlers are submitted to the schedule, AUA acts in the same way DASA acts. This leads directly to Theorem 1.

Theorem 1. *If a set of threads, T_r, having no handlers associated with it, is independently scheduled by DASA and AUA, the resulting schedules will be identical.*

Proof. AUA calculates the PUD of a thread in the same way DASA does. AUA analyzes threads in the same decreasing PUD order as DASA. AUA uses the same feasibility criteria as DASA. AUA orders feasible threads using EDF, just as DASA does. Because the two algorithms only differ when handlers are present, theorem 1 holds true. □

Algorithm 2: AUA UA optimization

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

If σ is not empty, the setLCT() function sets a timer that will wake the scheduler when the next LCT arrives. The dependencies and utility density (UD) of each thread are then calculated using getDep() and computeUD(), respectively. Dependencies and utility densities are calculated in AUA in the same way they are calculated in DASA. The function sortByUD()

then uses the utility density to create a list of all the threads in the system sorted in non-increasing order by UD. AUA goes through this list in order and attempts to insert each thread into a feasible, deadline ordered list using procedure insertByEDF(). Finally, AUA returns the thread T_{exe}, which is the task within the feasible schedule with the earliest deadline.

The computeUD() function sums the utilities of a thread and its dependencies. The function then divides the sum of utilities by the sum of execution times for a thread and its dependencies. This yields the aggregate utility density the system can expect from executing the thread and all threads upon which it depends. Utility density is used as a heuristic for selecting the most valuable tasks to execute. The getDep() function returns the thread that holds the resource that T_i is requesting.

The insertByEDF() function inserts a task and its dependencies into a deadline ordered list. Initially, the function makes a copy of the schedule and inserts the new thread, T_i, into the schedule copy. The dependency chain is then iterated through and each dependency's deadline is tightened before the dependency is added to the schedule copy. The function returns the copy of the schedule without making any changes to the original.

AUA's NBI is described in theorems 2 and 3.

Theorem 2. *The worst-case NBI of AUA is* $+\infty$.

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

Theorem 3. *The best-case NBI for AUA is zero.*

Proof. The best-case NBI for AUA occurs when a handler has been released, but it and all other accepted handlers are feasible with the newly arriving thread and its handler. In this

case, the newly arriving thread will be immediately included in a feasible schedule, resulting in an NBI of zero. □

3.3 Distributable Deadlines

AUA can be used to enforce ordered cleanup. As described above, AUA keeps track of the expected execution times of all admitted cleanup handlers. Using this information, AUA has the ability to create a new, shorter deadline for each downstream section.

The AUA algorithm calculates the new deadline d_{new} such that, $d_{new} = d_{old} - \sum_{i=1}^{n} C_i$ where C_i is the worst-case execution time of an admitted task abort handler and n is the number of cleanup handlers admitted for the section. AUA is then able to pass this new deadline off to the distributable thread framework for propagation downstream.

Propagating a shorter deadline will cause all downstream sections to have a smaller LCT, forcing all downstream sections to begin cleanup before the upstream sections, thus ordered cleanup behavior is enforced.

Chapter 4

The HUA Algorithm

4.1 Rationale

Since the task model is dynamic—i.e., when threads will arrive at nodes, and how many sections a thread will have are statically unknown, future scheduling events[1] such as new thread arrivals cannot be considered at a scheduling event. Thus, section schedules must be constructed on the system nodes solely exploiting the current system knowledge. Since the primary scheduling objective is to maximize the total thread accrued utility, a reasonable heuristic is a "greedy" strategy at each node: Favor "high return" thread sections over low return ones, and complete as many of them as possible before thread termination times, as early as possible (since TUFs are non-increasing).

The potential utility that can be accrued by executing a thread section on a node defines a measure of that section's "return on investment." We measure this using a metric called the *Potential Utility Density* (or PUD) originally introduced in [5]. On a node, a thread section's

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

PUD measures the utility that can be accrued per unit time by immediately executing the section on the node.

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

Definition 2 (Section Failure)**.** *Consider a section S_i of a distributable thread T_i. We say that S_i has failed when (a) S_i violates the termination time of T_i while executing, thereby raising a time constraint-violation exception on S_i's node; or (b) a failure-exception notification is received at S_i's node regarding the failure of a section of T_i that is upstream or downstream of S_i.*

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

Definition 3 (Released Handler)**.** *A handler is said to be released for execution when its section fails according to Definition 2.*

Since a section's best-case failure scenario is the absence of a failure for the section, the corresponding section PUD can be obtained as the utility accrued by executing the section divided by the time spent for executing the section. The section PUD for the worst-case failure scenario (one where the section fails, per Definition 2) can be obtained as the utility accrued by executing the handler of the section divided by the total time spent for executing the section and the handler.[2] The section's PUD can now be measured as the minimum of these two PUDs, as that represents the worst-case.

Thus, on each node, HUA examines thread sections for potential inclusion in a feasible schedule for the node in the order of decreasing section PUDs. For each section, the algorithm examines whether that section and its handler can be feasibly completed (we discuss section

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

and handler feasibility in section 4.2). If infeasible, the section and its handler are rejected. The process is repeated until all sections are examined, and the schedule's first section is dispatched for execution on the node.

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

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

This process ensures that the sections that are included in a node's schedule at any given time have feasible handlers. Further, all the upstream sections of their distributable threads also have feasible handlers on their respective nodes. Consequently, when any such section fails (per Definition 2), 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).

4.2 Feasibility

Feasibility of a section on a node can be tested by verifying whether the section can be completed on the node before the section's distributable thread's end-to-end termination time. Using a thread's end-to-end termination time for verifying the feasibility of a section of the thread may potentially overestimate the section's slack, especially if there are a significant number of sections that follow it in the thread. However, this is a reasonable choice, since we do not know the total number of sections of a thread. If the total number of sections of a thread is known a-priori, then better schemes (e.g., [19]) that distribute the thread's total slack (equally, proportionally) among all its sections can be considered.

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

There is always the possibility that a new section S_i is released on a node after the failure of another section S_j at the node (per Definition 2) and before the completion of S_j's handler on the node. As per the best-effort philosophy, S_i must immediately be afforded the opportunity for feasible execution on the node, in accordance with its potential contribution to the total utility. However, it is possible that a schedule that includes S_i on the node may not include

S_j's handler. Since S_j's handler cannot be rejected now, as that will violate the commitment previously made to S_j, the only option left is to not consider S_i for execution until S_j's handler completes, consequently degrading the algorithm's best-effort property. In Section 4.4, we quantify this loss.

4.3 Algorithm Overview

HUA's scheduling events at a node include the arrival of a thread at the node, completion of a thread section or a section handler at the node, and the expiration of a TUF termination time at the node. To describe HUA, we define the variables and auxiliary functions used in tables 4.1 and 4.2, respectively.

Table 4.1: Variables in HUA

$\mathcal{S}_r$	the current set of unscheduled sections including any newly arrived sections
S_i	a section within the system
S_i^h	handler for S_i
T_i	the thread to which a section S_i and S_i^h belong
σ_r	the EDF-ordered schedule constructed at the previous scheduling event
σ	the new EDF-ordered schedule
$U_i(t)$	S_i's TUF, which is the same as that of T_i's TUF
$U_i^h(t)$	S_i^h's TUF
$S_i.X$	S_i's termination time, which is the same as that of T_i's termination time
$S_i.ExecTime$	S_i's estimated remaining execution time
H	the set of handlers that are released for execution on the node (per Definition 3), ordered by non-decreasing handler termination times. $H = \emptyset$ if all released handlers have completed

Algorithm 3 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

Table 4.2: Operations used in HUA

`updateHandlerSet()`	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.
`notifyFail(`S_i`)`	declares S_i as failed .
`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, then 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.
`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 $S.ExecTime$.

predicted completion time exceeds its termination time, it is rejected (line 3). Otherwise, HUA calculates the section's PUD (line 3). To compute a section's PUD, HUA determines the PUDs for the best-case and worst-case failure scenarios and determines the minimum of the two.

The sections are then sorted by their PUDs (line 3). In each step of the *for*-loop from line 3 to line 3, the section with the largest PUD and its handler are inserted into σ, if it can produce a positive PUD. The schedule σ is maintained in the non-decreasing order of section termination times. Thus, a section S_i is inserted into σ at a position that corresponds

to $S_i.X$ in σ's non-decreasing termination time order. S_i^h is similarly inserted into σ at a position corresponding to $S_i.X + S_i^h.X$.

Algorithm 3: HUA: High Level Description

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

After inserting a section S_i and its handler S_i^h, the schedule σ is tested for feasibility. If σ becomes infeasible, then S_i and S_i^h are removed from σ (lines 3–3). If a section S_i that is removed from σ is not a head and belonged to the schedule constructed at the previous scheduling event, then the TMAR protocol is notified regarding S_i's failure (lines 3–3).

If one or more handlers have been released but have not completed their execution (i.e., $H \neq \emptyset$; line 3), the algorithm checks whether any of those handlers are missing in the schedule σ (lines 3– 3). If any handler is missing, the handler at the head of H is selected for execution (line 3). If all handlers in H have been included in σ, the section at the head of σ is selected (line 3).

Note 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. Actually, we are calculating the highest possible PUD of each section by assuming that it is executed at the current position. Intuitively, this would benefit the final PUD, since the section with the highest PUD is always selected at each insertion on σ (line 3). Also, the PUD calculated for the dispatched section at the head of σ is always accurate.

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

4.4 Algorithm Properties

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

Theorem 4. *If a section S_i fails (per Definition 2), then under HUA with zero overhead, its handler S_i^h will complete no later than $S_i.X + S_i^h.X$ (barring S_i^h's failure).*

Proof. If S_i violates the thread termination time at a time t while executing, then S_i was included in HUA's schedule constructed at the scheduling event that occurred nearest to t,

say at t', since only threads in the schedule are executed. Thus, both S_i and S_i^h were feasible at t', and S_i^h was scheduled to complete no later than $S_i.X + S_i^h.X$. Similar argument holds for the other cases:

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

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

Next, we describe HUA's NBI:

Theorem 5. *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*.*

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, 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 $t + \max_{\forall S_j \in \sigma_t} \left(S_j.X + S_j^h.X\right)$.

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

Theorem 6. *The best-case NBI for HUA is zero.*

Proof. The best-case NBI for HUA occurs when a handler has been released, but it is feasible with the newly arriving thread and its handler. In this case, the newly arriving thread will be immediately included in a feasible schedule, resulting in an NBI of zero. □

HUAproduces optimum total utility for a special case:

Theorem 7. *Consider a set of threads with step TUFs and no node failures. Suppose there is sufficient processor time for meeting the termination-times of all thread sections and their handlers on all nodes. Now, a system-wide EDF schedule is produced by HUA, yielding optimum total utility.*

Proof. Because each thread is inserted into a deadline-ordered list by HUA the resulting schedule will be an EDF schedule. □

Chapter 5

The TPR Protocol

5.1 Assumptions

The Thread Polling with Bounded Recovery(TPR) protocol was designed to effectively monitor thread integrity of distributed threads within a relatively reliable network. As such, TPR focuses less on recovering from communication errors and is more concerned with identifying and recovering from catastrophic node failures. TPR provides tunable parameters to mitigate the effects of unreliable communications, but it was not designed for efficient operation in such an environment. TPR assumes a reliable communication framework for administrative messages with a maximum latency of D.

5.2 Overview

The TPR protocol is an extension of the Alpha TMAR protocol described in [14] and is instantiated in a software component called the *Thread Integrity Manager* (TIM). Every

node which hosts distributable threads has a TIM component, which continually runs TPR's three-phase polling operation.

The TIM on each node is responsible for maintaining the health of each segments locally hosted, determining the integrity of all threads rooted locally, and coordinating any cleanup required for those threads. A TIM responsible for a non-root segment, then, manage the segment's health by responding to health update information sent by the root TIM. If health information fails to arrive for a given amount of time, the local TIM declares the segment an *orphan* and the segment commences autonomous cleanup. Once this occurs, the thread segment is effectively disconnected from the remainder of the thread's distributed call-graph and stack (as described earlier in section 2.5, and control is returned to application code in the context of an exceptional cleanup handler.

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

5.3 Thread Polling

In the first phase, the root node of a given thread regularly broadcasts an ROOT_ANNOUNCE message to all nodes within the system. The ROOT_ANNOUNCE message is sent every T_p, or polling interval. Figure 5.1 illustrates the polling process for a healthy thread.

In the second phase, all nodes that are hosting segments of that given thread respond to the ROOT_ANNOUNCE with a segment acknowledgment (SEG_ACK) message. Table 5.3 describes the contents of the SEG_ACK and other messages used in TPR.

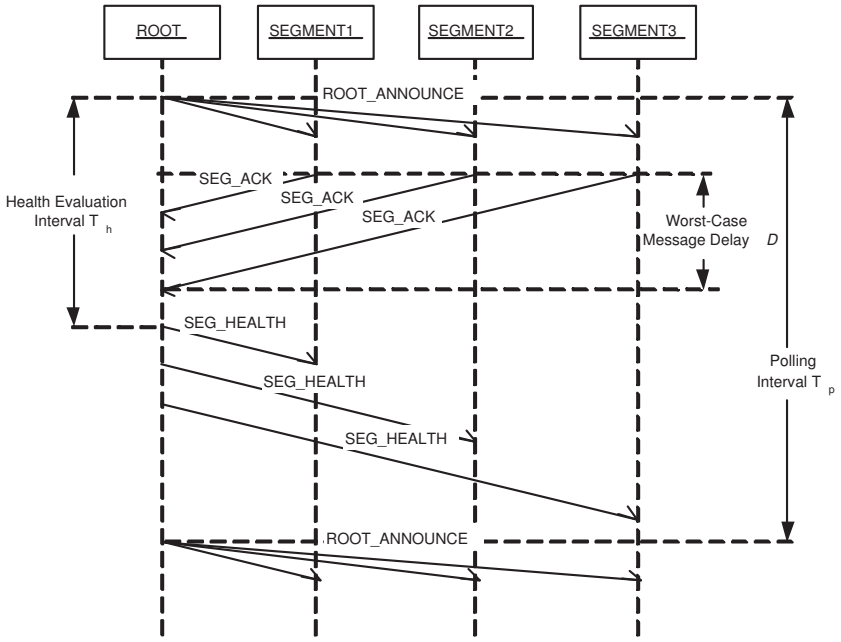

Figure 5.1: TPR Operation — Healthy Thread

Lemma 8. *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 last phase, the root node waits for the health evaluation interval T_h to expire before examining the information it has received from the SEG_ACK messages to determine the status of the thread (broken or unbroken).

Lemma 9. *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 imme-

Table 5.1: TPR Messages

Message	Contents	From/To
ANNOUNCE	list of all DTs rooted on sender node	root node to all nodes in system
SEG_ACK	list of all segments located on sender that are parts of DTs rooted on receiver	all nodes with segments of DT to DT root node
SEG_HEALTH	list of all healthy segments located on receiver	DT root node to nodes with healthy segments of DT
PAUSE	ID of DT to be paused	root node to all nodes in system
PAUSE_ACK	ID of DT that was paused	head node to root node
UNPAUSE	ID of DT to be unpaused	root node to all nodes in system
NEW_HEAD	segment ID of new head	root node to node with new head
KILL_SEG	segment ID of old head	root node to node with old head
KILL_ACK	ID of DT whose head was orphaned	node with old head to root node
ORPHAN	list of all segments to be orphaned on a node	root node to a node hosting orphans
ORPHAN_HEAD	segment ID of orphan head	root node to node hosting furthest downstream segment of unbroken piece of thread

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

Lemma 10. *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. □

5.4 Recovery

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

5.4.1 Recovery Modes

Recovery can be conducted in one of two modes. The first mode is single-head mode and the second is muli-head mode. As their names imply, single-head mode allows at most one head for a given distributed thread and muti-head mode allows for more than one head to exist at a time. The important distinction between the two is that single-head recovery provides asynchronous, autonomous recovery for thread segments while multi-head recovery allows for synchronous, ordered cleanup. This difference will be expounded upon in the TPR Recovery Process section.

5.4.2 Recovery Process

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.

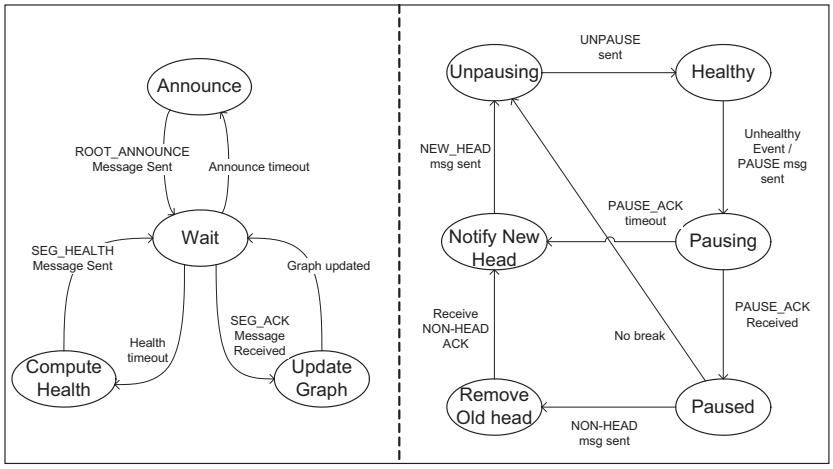

Figure 5.2: High-level State Diagram — Root Segment (Single-head)

Figure 5.2 illustrates the states experienced by an individual thread during single-head recovery from the standpoint of its root segment. The state of the TPR protocol at any given time can be described as the set of one state from the left of the figure and one state from the right. The left half of the figure represents the states used for the polling mechanism and the right half of the figure represents the states used for recovery. When the protocol is not recovering from a perceived error, the recovery state is "Healthy".

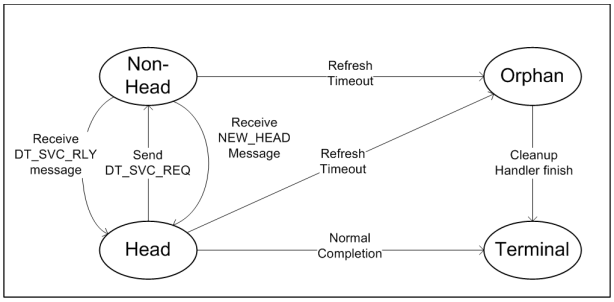

Figure 5.3: High-level State Diagram — Section

Figure 5.3 illustrates the states a section will traverse under the TPR protocol. These are separate from the protocol/root states shown in figure 5.2.

In the first step (see Figure 5.4), the recovery operation broadcasts a PAUSE message and waits. The recovery thread continues waiting until it either receives a PAUSE_ACK message from the current head of the thread or a user-specified amount of time lapses without a PAUSE_ACK message being received. In the second step, the recovery operation analyzes the thread's distributed call-graph and finds the farthest contiguous thread segment from the root. This segment will be the new head. If the old head still exists after 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.

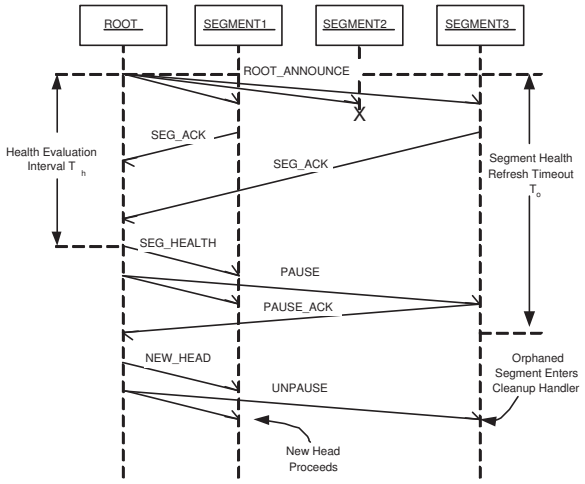

Figure 5.4: TPR Operation — Unhealthy Thread Entering Recovery

Lemma 11. *Once a thread failure is detected, TPR activates a new thread head within* $t_p + t_h + 4D$.

Proof. By Lemma 9, 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. □

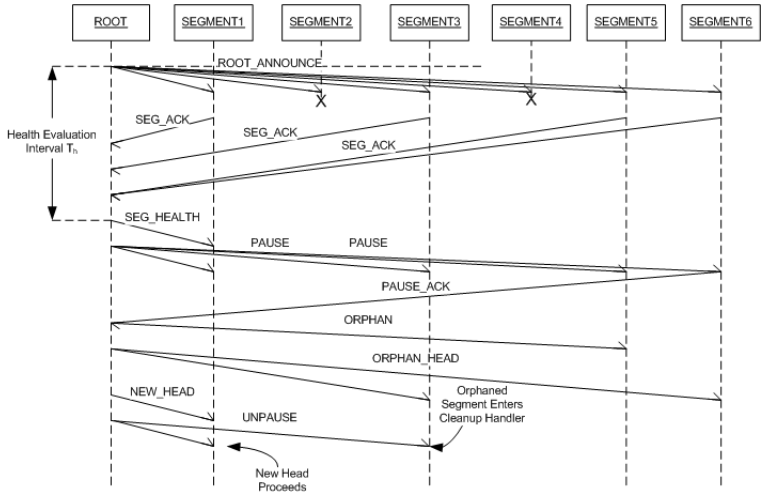

Figure 5.5: TPR Operation — Unhealthy Thread Entering Recovery (muli-head)

In multi-head recovery, the protocol performs steps (1) and (2) in the same manner as single-head recovery (see figure 5.5). However, multi-head recovery does not wait for acknowledgement that the old head has been cleaned-up before proceeding (see Figure 5.6).

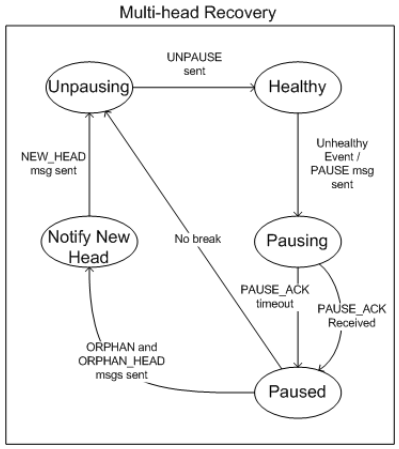

Figure 5.6: High-level State Diagram — Root Segment (Multi-head)

Instead, the protocol sends ORPHAN messages to all segments no longer connected to the root in the thread's call-graph. This ensures that when the old head has finished cleanup,

its preceding segments will be ready to begin their own cleanup. If multiple breaks have occurred, ORPHAN_HEAD messages will be sent to all segments (excluding the new head segment) directly preceding a break Figure 5.7 shows where the messages will be delivered for this specific break ('x' denotes ORPHAN_HEAD and 'o' ORPHAN). When a segment receives the ORPHAN_HEAD message, it will begin execution of cleanup code. As a result, a single thread may have several *heads* performing cleanup in addition to the true head of the thread, which is performing normal task execution; hence the name *multi*-head recovery.

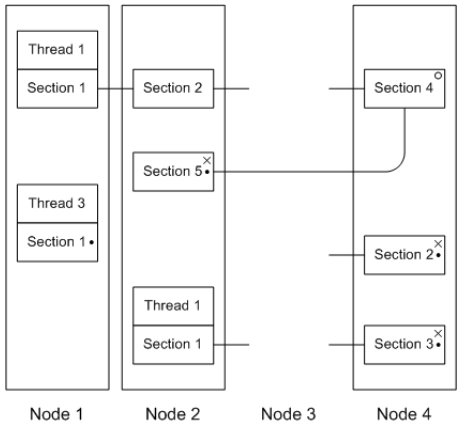

Figure 5.7: Delivery locations of ORPHAN and ORPHAN_HEAD messages

Multi-head recovery mode allows TPR to perform timely, ordered cleanup in the face of multiple node failures. TPR's single-head mode of recovery only specifies a single point of execution to perform cleanup activities. This single point of execution then propagates itself back towards the root and stop when it reaches the thread break. In the case of multiple breaks, segments in between breaks will not begin cleanup until their respective LCTs. The multiple points of cleanup execution allow TPR to avoid this situation.

Once the new head has been notified, the point of execution returns to application code exactly where the new head made its last remote invocation. An exception is returned to indicate that a thread integrity failure has occurred, and it is the responsibility of the application programmer to determine how the application should handle the exception.

5.5 Orphan Cleanup

When a segment has not been refreshed for a specified amount of time, the segment is flagged as an orphan, ready for cleanup. If the protocol is in single-head mode, the orphan will begin cleanup immediately. If the protocol is in multi-head mode, the orphan will begin cleanup only if it is an *orphan-head*. Otherwise, the orphan will wait to begin cleanup until its successor segment tells it to begin.

Until the LCT of abort handlers, cleanup code is scheduled and run in the same way any application code is run, meaning that cleanup code is subject to interference from other application threads up until the handler's LCT has been reached. This pushes the cleanup bound all the way out to the thread's deadline.

Orphan cleanup serves both to remove segments that follow a break in the distributable thread (called *thread trimming*) and to remove the entirety of threads that have lost their root.

Theorem 12. *If threads are scheduled using* HUA *or* AUA *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 10 implies that every unhealthy section S_i will detect that it is an orphan within $t_p + t_h + D$. Theorem 4 implies that S_i's handler will complete within $S_i.X + S_i^h.X$, once S_i fails per Definition 2. Definition 2 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. □

5.5.1 Ordered Cleanup

In conjunction with AUA ordered cleanup may be accomplished using single-head recovery, but it requires that the timing constraints be appropriately tuned to the network the application will be running on. Each orphan in a distributable thread will be blocked until it either receives notification from its downstream successor segment or its LCT arrives. When multiple breaks occur it is impossible for the segment to receive notification from its successor. Therefore, the only way for single-head recovery to perform ordered cleanup in a multiple break situation is by supplying the appropriate timing constraints to AUA.

Theorem 13. *Threads maintained using TPR in multi-head recovery mode will perform ordered cleanup when cleanup is necessary.*

Proof. As TPR with muti-head recovery ensures that the only points of execution are those furthest downstream for any given *piece* of a thread, ordered cleanup is assured. □

For multi-head recovery, ORPHAN messages are sent to all orphaned segments preparing them for cleanup. Cleanup is only allowed to begin with the delivery of an ORPHAN_HEAD message, which is only delivered to the segment that is farthest downstream in a thread *piece*. Therefore, cleanup within a *piece* can only begin on the farthest downstream segment, the orphan-head. Once cleanup has finished on the the orphan-head, its point of execution is transferred upstream to the orphan-head's predecessor via a remote return. As communications are assumed to be reliable except in the case of node failure, cleanup is guaranteed to continue propagating upstream through the entire thread *piece*, which complies with the definition of ordered cleanup specified at the end of Chapter 2.

Chapter 6

The D-TPR Protocol

6.1 Overview

There were no assumptions made on the reliability of the communication network while designing this TMAR protocol. Thus, the Decentralized Thread Polling with Bounded Recovery TMAR protocol (D-TPR) is able to effectively perform recovery in a wide variety of environments.

D-TPR was designed to satisfy the need for a light-weight, decentralized TMAR protocol. It is based on Thread Polling as described in [6, 27], but instead of the root being responsible for identifying breaks, pair-wise communication between predecessor and successor nodes allow the protocol to determine if and where a break occurs in a more decentralized manner.

Figure 6.1 shows a sequence diagram for the healthy operation of D-TPR. It an be seen that D-TPR has several characteristics in common with TPR and the original Thread Polling (e.g. polling cycles, orphan timeouts, etc.)

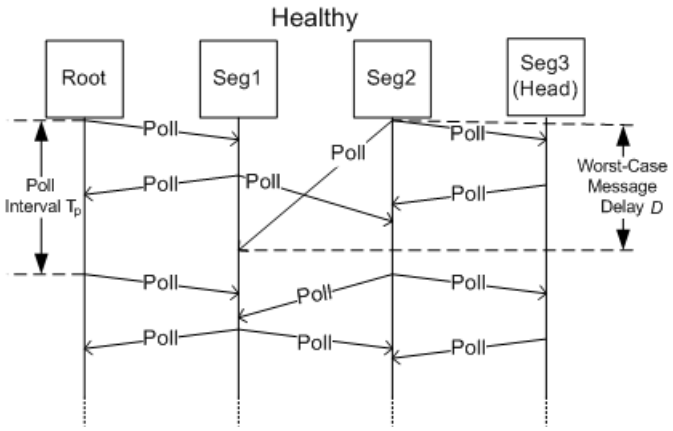

Figure 6.1: D-TPR Operation — Healthy Thread

6.2 Polling

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

Table 6.1: D-TPR Messages

Message	Contents	From/To
POLL	List of local segmentID and remote segmentID pairs. Remote segmentIDs are either predecessor or successor segments to local segment	travel back and forth between predecessor and successor nodes
NEW_HEAD	timed out segment and predecessor segment	node with upstream timeout to predecessor node
ENDORPHAN	timed out segment and successor segment	node with downstream timeout to successor node
ORPHANPROP	orphaned segment and successor segment	node with orphan segment to successor node

Each POLL message (see Table 6.2) is a list of entries, where each entry contains a type, the local segment ID the entry corresponds to, and a remote segment ID. If the entry type

is SUCCESSOR, the remote segment ID will correspond to the successor segment of the local segment in the entry. Similarly, the remote segment ID of PREDECESSOR corresponds to the predecessor segment of the local segment in the entry. In this way, the node receiving the POLL message is able to discern (downstream or upstream) the message's origin and thus from which direction the segment has been deemed healthy. This distinction becomes important for break detection and is discussed further in the following section.

6.3 Break Detection

When an invocation is made, the protocol creates two timers set to application specified timeouts. One timer is established for the downstream segment and the other is established for the upstream segment. The TIM on the node making the invocation (upstream side) creates a downstream-invocation timer that will cause a timeout when polling messages have not been received from downstream frequently enough. The TIM on the node hosting the remote object to which the invocation is being made (downstream side) creates an upstream-invocation timer that will cause a timeout when polling messages are not received from upstream frequently enough.

When a POLL message is received from upstream, the upstream-invocation timer is reset to some application specific time and resumes counting down. The same is true of the downstream-invocation timer when a POLL message is received from downstream.

A "break" is declared when either the upstream or downstream-invocation time reaches zero. Recovery is different depending on which timer experiences the timeout

Lemma 14. *Consider a section S_i and its successor section S_j. Under D-TPR, if S_j's node fails, or S_i becomes unreachable from S_j (but not necessarily vice versa), then S_i will detect a thread break between S_i and S_j within $t_p + D$.*

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

Lemma 15. *Consider a section S_j and its predecessor S_i. Under D-TPR, if S_i's node fails, or S_j becomes unreachable from S_i (but not necessarily vice versa), then S_j will detect a thread break between S_i and S_j within $t_p + D$. S_j and its downstream sections are now said to be orphaned.*

Proof. The proof is similar to that of Lemma 14. □

6.4 Recovery

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

In order to force the upstream segment to become the *new head*, the protocol sends a NEW_HEAD message upstream and ceases upstream POLL messages, which refresh the upstream segment. If the upstream node receives the NEW_HEAD message, the upstream segment will immediately begin behaving like a *new head*. If the upstream node does not receive

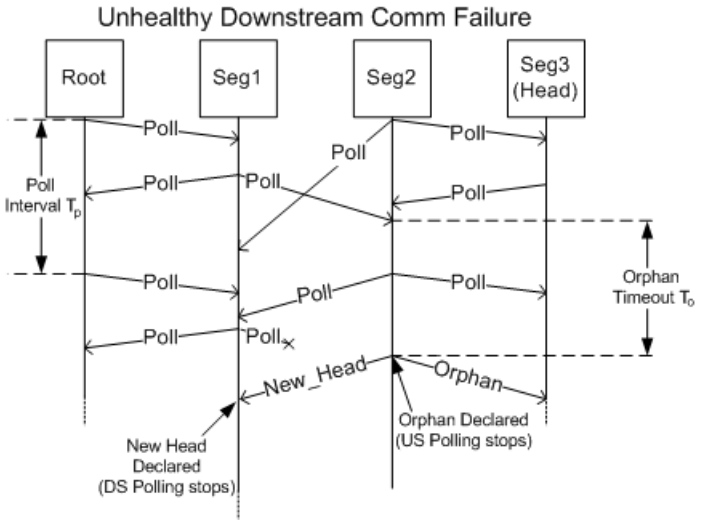

Figure 6.2: D-TPR Operation — Unhealthy Thread

the message, the upstream segment's downstream-invocation timer will expire (due to the stopped POLL messages) forcing the segment to become the *new head*.

In order to force the downstream segment to become an *orphan*, the protocol sends an ORPHANPROP message downstream and modifies its downstream POLL messages to include an orphan status. The downstream node will either receive the ORPHANPROP message and become an *orphan*, or the downstream segment's timer will expire forcing it to become an *orphan*. When a segment becomes an orphan it propagates the ORPHANPROP message in order to have all orphans correctly identified more quickly.

When a segment's downstream-invocation timer expires, the protocol assumes that the downstream segments are unreachable and declares itself the *new head* of the thread. The *new head* then sends an ENDORPHAN downstream and ceases downstream refresh polling. In this way, the downstream segment will either receive the ENDORPHAN notification and become and *orphan* or it's upstream timer will expire, making the segment an orphan.

Lemma 16. *Under D-TPR, if a thread break occurs between S_i and its successor S_j, then S_i will become the new head within $t_p + 2D$. Since the new head of a thread is always directly*

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

Proof. A thread break between S_i and S_j can occur in primarily two ways: (I) S_j's node fails, or S_i becomes unreachable from S_j; and (II) S_i's node fails, or S_j becomes unreachable from S_i. Lemma 14 identifies Case (I). Thus, S_i will detect the thread break within $t_p + D$, and immediately after, S_i will declare itself as the new head, within $t_p + D$. Case (II) is identified in Lemma 15. Thus, S_j will detect the break within $t_p + D$ and will send a NEW_HEAD message to S_i. Upon receipt of this message, S_i will declare itself as the new head, within a total of $t_p + 2D$ (after break detection), which is the worst-case. Lemma follows. □

Lemma 17. *Under D-TPR, if a thread break occurs between S_i and its successor S_j, then S_j will identify itself as an orphan within $t_p + 2D$.*

Proof. Proof is similar to that of Lemma 16. A thread break between S_i and S_j can occur in primarily two ways: (I) S_i's node fails, or S_j becomes unreachable from S_i; and (II) S_j's node fails, or S_i becomes unreachable from S_j. Case (I) is identified in Lemma 15. Thus, S_j will detect the break within $t_p + D$ and will immediately declare itself as an *orphan*, within $t_p + D$. Case (II) is identified in Lemma 14. Thus, S_i will detect the thread break within $t_p + D$, declare itself as the new head, and send an ENDORPHAN message to S_j. Upon receipt of this message, S_j will declare itself as an *orphan*, within a total of $t_p + 2D$ (after break detection), which is the worst-case. Lemma follows. □

6.5 Cleanup

An *orphan* begins executing abort code only if it has been granted execution permission by being designated an orphan-head. This can happen in one of three ways:

1. The current head of the thread becomes an orphan;

2. A non-head orphan is returned to by an orphan-head and becomes a new orphan-head;

3. An orphan's downstream-invocation timer expires forcing it to become a new orphan-head.

Theorem 18. *Under D-TPR/HUA, if a thread break occurs between a section S_i and its successor S_j, then all orphans from S_j till the thread's current head S_{j+k}, for some $k \geq 1$, will be aborted in the LIFO-order—i.e., from S_{j+k} to S_j—and will complete by $t_p + (2 + k)D + \sigma_{\alpha=0}^{k}(S_{j+\alpha}.X + S_{j+\alpha}^{h}.X)$, unless a section $S_{j+\alpha}$ becomes unreachable from $S_{j+\alpha+1}, 0 \leq \alpha \leq k-1$.*

Proof. Let the thread's execution sequence be: $\langle \cdots S_i, S_j, S_{j+1}, \cdots, S_{j+k} \rangle$. From Lemma 17, S_j will identify itself as an orphan within $t_p + 2D$. Following this, the ORPHANPROP message will be propagated from S_j to S_{j+k} within kD. Thus, S_{j+k} will become the first orphan-head and thus the first orphan to be aborted, followed by S_{j+k-1}, S_{j+k-2}, until S_j, following the LIFO-order, since $S_{j+k-\alpha}$ is always returned to by $S_{j+k-(\alpha-1)}, 0 \leq \alpha \leq k$ by the thread's execution sequence.

By Theorem 4, a section S_α's handler will complete within $S_\alpha.X + S_\alpha^h.X$, once it is an orphan-designate. Thus, all sections from S_j to S_{j+k} will complete within $t_p + 2D + kD + \sigma_{\alpha=0}^{k}(S_{j+\alpha}.X + S_{j+\alpha}^{h}.X)$. Theorem follows.

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

Theorem 19. *, Under D-TPR/HUA, if a thread breaks, then the thread's orphans will complete within a bounded time.*

Proof. This theorem follows from Theorem 18, except for the case when a section $S_{j+\alpha}$ becomes unreachable from $S_{j+\alpha+1}$ $(0 \leq \alpha \leq k-1)$ after a break occurs between S_i and its successor S_j. If $S_{j+\alpha}$ becomes unreachable from its successor $S_{j+\alpha+1}$, then $S_{j+\alpha}$'s downstream invocation timer will expire within $t_p + D$ (similar to Lemma 14, where $S_i \equiv S_{j+\alpha}$ and $S_j \equiv S_{j+\alpha+1}$), designating $S_{j+\alpha}$ as orphan-head. By Theorem 4, now $S_{j+\alpha}$ will cleanup within $t_p + D + S_{j+\alpha}.X + S^h_{j+\alpha}.X$. Theorem follows. □

Chapter 7

The W-TPR Protocol

7.1 Assumptions

W-TPR is designed to provide bounded maintenance and recovery for Distributed Threads within a Mobile Ad-Hoc Network (MANET). It is assumed that communications in a MANET are unreliable and may be prone to transient failures. Therefore, W-TPR is designed to be more robust against these types of failures than the other TMAR protocols discussed in this thesis.

W-TPR exploits the fact that a thread is only adversely affected by a thread break if the head attempts to move across that break. The other TMAR protocols discussed in this thesis detect a break and assume the break will be permanent so they preempt the possibility of the head crossing the break by eliminating segments beyond the break point. W-TPR assumes that breaks are not permanent.

Because the underlying network is unreliable, it is assumed that W-TPR will be employed with a distributable thread framework which provides some reliability measures. In the case

of DRTSJ, W-TPR is provided with the ability to modify the rate and duration for resending invocation requests and return messages.

7.2 Description

W-TPR is similar to D-TPR in the fact that they are both decentralized protocols designed for mobile ad-hoc networks. W-TPR differs from the D-TPR protocol primarily in the way thread-breaks are determined. In D-TPR, breaks are recognized when communication between two consecutive nodes of a thread fails for longer than some application specified threshold time. In W-TPR, breaks are never actually recognized. Instead, W-TPR recognizes when communication errors affect either an invocation or a return (head movement) and provides maintenance accordingly.

Figure 7.1 shows the states and state transitions a segment can expect to go through in the W-TPR protocol. It is worthwhile to note that no breaks are ever declared and that a segment becomes an orphan only if it receives the ORPHAN message from an upstream segment. Segments assume they are healthy until notified otherwise.

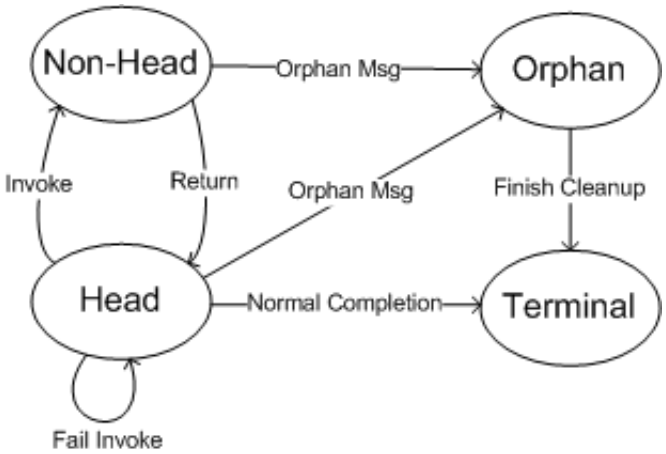

Figure 7.1: W-TPR Segment State Diagram for

7.2.1 Downstream Head Movement

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

When an invocation is made, an invocation request is sent downstream and the local segment, S_i, begins waiting for invocation verification. This verification can be given in two ways: the DT framework could notify the local segment of a successful invocation or the local segment could receive a POLL message from the remote node containing the segment ID of the remote segment. When the invocation is verified, the local segment is stopped until the remote segment performs a return (head moves downstream). Figure 7.2 shows an example of a successful invocation.

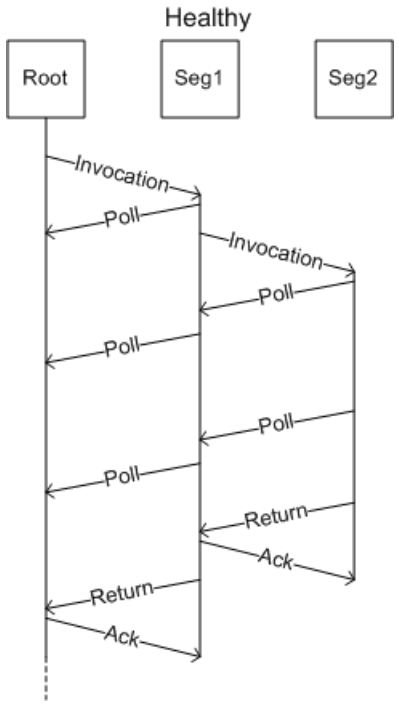

Figure 7.2: W-TPR Operation — Healthy Thread

Figure 7.3 shows an unhealthy attempt at an invocation caused by an upstream failure on

the left and a downstream failure on the right.

When the invocation is received by the downstream node, the downstream node attempts to finalize the invocation and begins sending periodic POLL messages to the upstream segment. When a healthy segment receives a POLL message from an *orphan*, the healthy segment returns an ORPHAN message to the *orphan*. If the *orphan* is not the *orphan-head*, the ORPHAN message is propagated upstream. All W-TPR messages are discribed in detail in Table 7.2.1.

Table 7.1: W-TPR Messages

Message	Contents	From/To
POLL	segment ID pair of S_i and S_{i-1}	downstream node to upstream node
RETURN-ACK	segment ID of segment attempting return	upstream node to downstream node after return success
ORPHAN	segment ID pair of S_i (healthy) and S_{i+1} (orphan)	

The protocol resends the invocation request until either the invocation is verified or the protocol deems that communication with the downstream node is not possible. If communication with the downstream node is not possible, the local segment maintains head status and the application is notified that the invocation failed. The TIM also sends an ORPHAN message downstream in the event that a partial invocation was accomplished. Figure 7.3 shows an unhealthy attempt at an invocation caused by an upstream failure on the left and a downstream failure on the right.

Lemma 20. *Under W-TPR, the location of a thread's head is ambiguous for at most* t_n.

Proof. Directly follows discussion. □

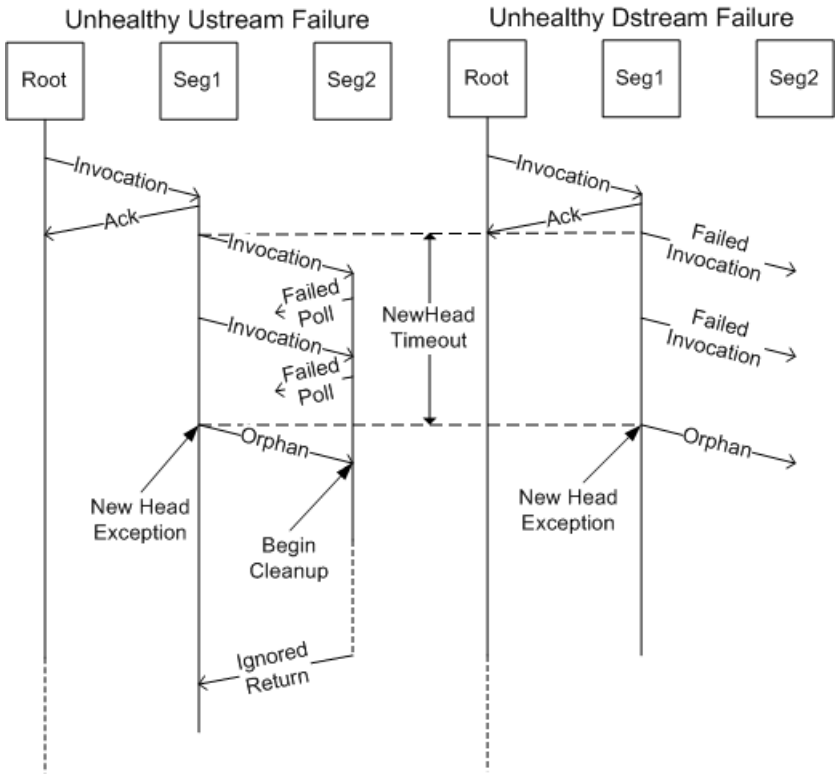

Figure 7.3: W-TPR Operation — Unhealthy Invocation

7.2.2 Upstream Head Movement

When the head is moving from the local node to an upstream node (i.e. remote return), the local node begins waiting for return verification from the upstream node. When the return message is received by the upstream node, the upstream node sends a return verification message downstream to the local node. If the verification is not received within T_r, the return timeout (see Figure 7.4), the protocol forces the return message to be resent. This process is repeated until the handshake successfully completes or the segment on the local node violates its timing constraint. Even in the presence of upstream communication errors, the downstream segment never becomes an *orphan*. Since the segment has already finished executing and has a healthy return value, it would be fruitless to abort this segment before delivering its return value.

Lemma 21. *Under W-TPR, a thread's head is never disconnected from the rest of the thread and no new head activation is required.*

Proof. Follows directly from the previous discussion. By Lemma 20, after t_n, the head moves downstream after a fully successful invocation. Any fully successful invocation can execute a return. If the upstream node becomes unreachable when the downstream node executes a return, the downstream section has completed its execution (hence it is returning) and is therefore not an *orphan*. □

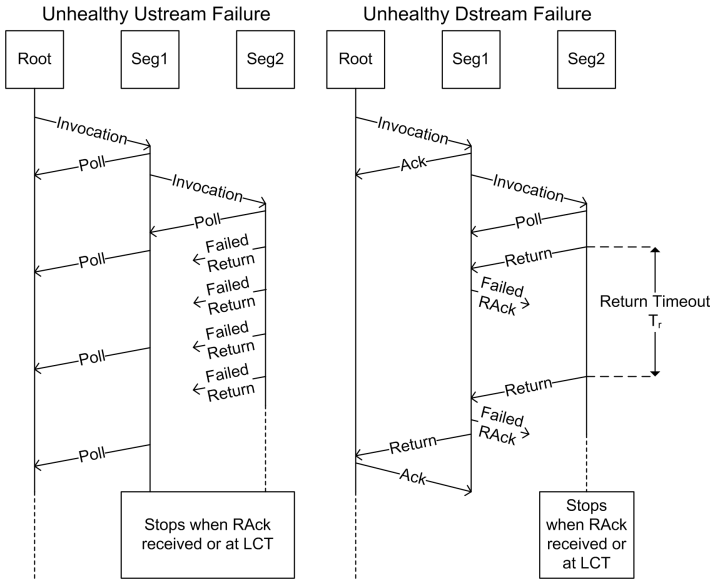

Figure 7.4: W-TPR Operation — Unhealthy Return

7.2.3 Cleanup

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

Theorem 22. *Under W-TPR if a section S_i makes an unsuccessful invocation to its (potential) successor section S_j (i.e., S_j will be S_i's successor had if the invocation was successful), then all orphans that can potentially be created from S_j till the thread's furthest orphaned section $S_{j+k}, k \geq 1$, will be aborted in the LIFO-order and will complete within a bounded time under HUA, as long as no further failures occur between S_j and S_{j+k}.*

Proof. By Lemma 20, after t_n, S_i retains the head status since the invocation was unsuccessful, and an ORPHAN message is propagated to all downstream sections till S_{j+k}. The rest of the proof follows that of Theorem 19. □

Note that Theorem 22 holds only if no further failures occur between S_j and S_{j+k}. If such a failure were to occur, then the ORPHAN message may not be propagated or an orphan-head may not be able to return to a non-head orphan. D-TPRcan detect such failures due to its continuous pairwise polling operation, whereas W-TPR is unable to do so precisely due its "on-demand" polling approach.

Theorem 23. *, Under W-TPR/HUA, if orphans are created for a thread as in Theorem 22, then all the orphans will complete within a bounded time, as long as no further failures occur between S_j and S_{j+k}.*

Proof. Follows from Theorem 22. □

Chapter 8

Experimental Evaluation

During the development of this project, two distributed thread frameworks were used for experimental evaluation. The initial framework was the *Tempus* middleware developed here at Virginia Tech by Peng Li.The second framework used was the DRTSJ reference implementation developed in part here at Virginia Tech by Jonathan Anderson.

Each Distributed Thread Framework (referred to herein simply as framework) used in this project required two basic capabilities. Each framework had to provide some way of performing thread propagation across a network and some kind of Case 2 (per node) scheduling mechanism.

The following sections describe each framework and provide the experimental results obtained from them.

8.1 Tempus

Tempus is a custom distributed middleware environment developed at Virginia Tech's Real-Time Systems Laboratory. This environment consists of an implementation of the distributable threads abstraction in the C programming language and a pluggable scheduling framework called the *Metascheduler* [22], which facilitates the composition of user-defined scheduling policies such as AUA.

Within *Tempus*, the Metascheduler and the DT framework became somewhat integrated allowing for the Metascheduler to provide support for scheduling situations unique to distributed threads–e.g, PAUSE. In Figure 8.1, 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 sections of a thread.

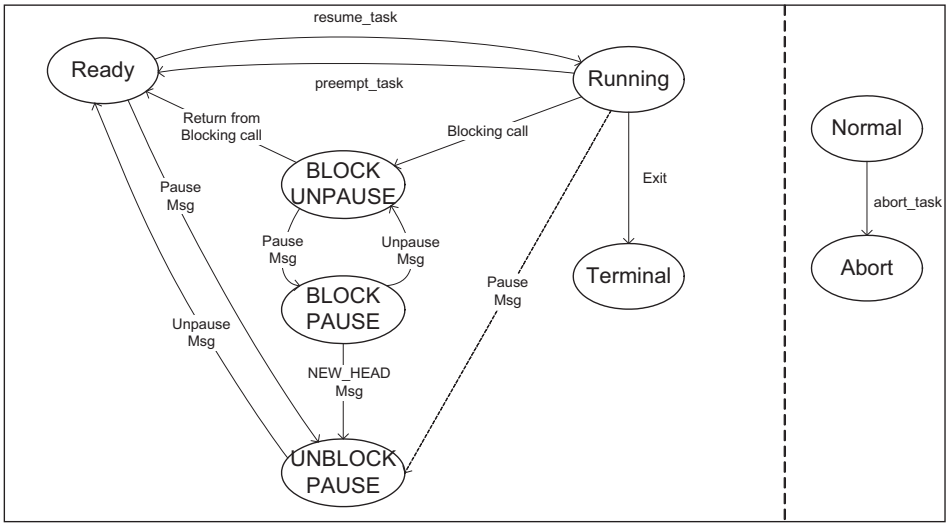

Figure 8.1: Thread Scheduling States

For this framework, only AUA and the TPR protocol were implemented.

8.1.1 Single Node Experiments

The experiments presented below were performed on a small testbed of Intel Pentium III-based PC's running QNX Neutrino 6.2.1. The interconnects consisted of commodity 10 megabit/sec interfaces on a switched Ethernet network. Each machine hosted an instance of the *Tempus* middleware and Metascheduler scheduling framework.

A number of experiments were carried out to establish the behavior of the AUA scheduling approach in a single node context. We measured the Accrued Utility Ratio (AUR), Deadline Satisfaction Ratio (DSR), and Deadline Miss Load (DML) produced by our implementation under a variety of load and task structure conditions.

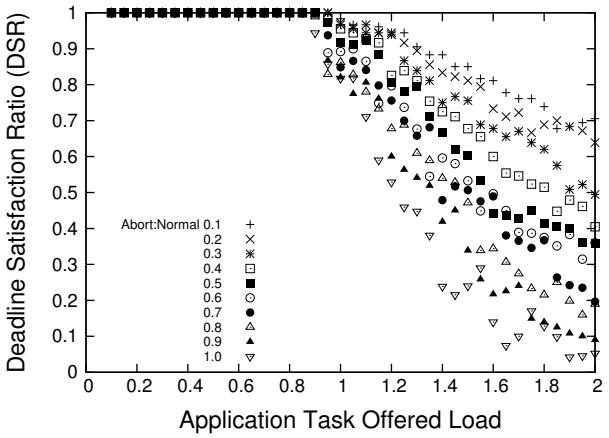

Figure 8.2: Deadline Satisfaction Ratio

Deadline Satisfaction Ratio: The *DSR* metric is defined as the number of tasks which complete by their deadline divided by the total number of tasks. *DSR* is therefore convenient for comparison to traditional deadline-driven scheduling approaches. As with our *AUR* measurements, we conducted experiments to profile deadline satisfaction over a range of load conditions. On the horizontal axis, the offered application task load is ramped from zero to 200% of available CPU capacity. Up to a certain load—when the system is "underloaded"—every deadline is satisfied. As the load increases beyond the "deadline miss load" (presented

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

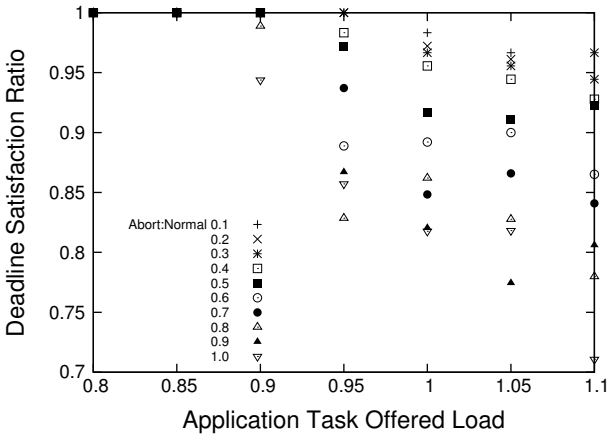

Figure 8.3: Deadline Satisfaction Ratio (detail)

Upon closer inspection (see Figure 8.3), it can be seen that the highest load at which AUA misses no deadlines is a function of the currently accepted load of abort handlers. Intuitively, this is the correct behavior since AUA effectively reserves schedule to ensure that cleanup handlers are feasible in the presence of any offered application load. The data show that AUA is nevertheless able to degrade gracefully as the load increases, continuing to meet significant fractions of the time constraints despite operating in overload.

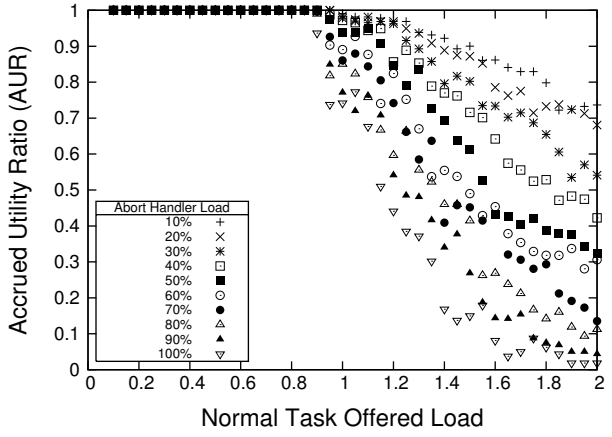

Figure 8.4: Accrued Utility Ratio

Accrued Utility Ratio: The AUR is defined as the ratio of utility earned by executing

tasks successfully divided by the total utility of all tasks in the offered load. This is a direct measurement of the "value" delivered to the application tasks. The data presented in Figure 8.4 illustrates the accrued utility as the offered load on the scheduler is elevated from 0 to 2.0. As we have argued above, AUA delivers a 1.0 accrued utility ratio—it satisfies the deadline of all tasks, irrespective of their utility—when operating in underload. This data bears out the claim that AUA is equivalent to DASA, and hence EDF, in underloads.

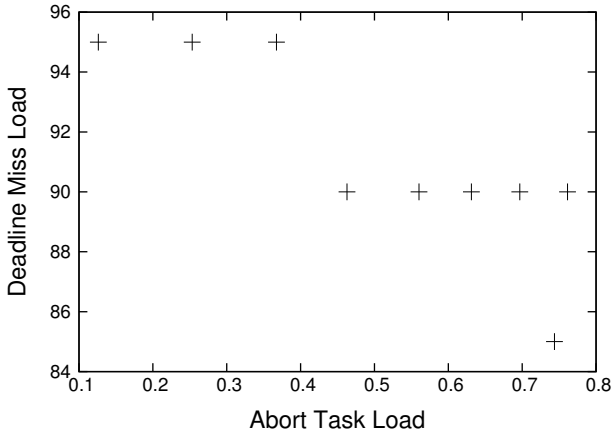

Figure 8.5: Deadline Miss Load

Deadline Miss Load: The *DML* of a scheduler is defined to be the offered load under which the scheduler begins missing task deadlines. Ideally, the *DML* would occur at precisely a load of 1.0; the scheduler would never miss a feasible deadline. Because of implementation-induced overhead such as context switch latency and time spent in scheduler and operating system code, it is not possible to achieve this theoretical maximum.

Furthermore, the overhead associated with scheduler and OS logic becomes more pronounced as task time constraints decrease, becoming very pronounced when the task execution times are on the same order as scheduling latencies. In addition, we show in Figure 8.5 that the *DML* is also adversely affected by the abort load induced by the currently-accepted set of threads. However, the algorithm performs reasonably well for low abort loads, missing no deadlines at 95% of theoretical capacity, despite a 30% load for abort reservations.

8.2 DRTSJ and the RTSJ-Metascheduler

The Distributed Real Time Specification for Java (DRTSJ) [1] reference implementation is an implementation of the distributable thread abstraction based on Real-Time Java and Java's Remote Method Invocation (RMI) mechanism. The RTSJ-Metascheduler is a scheduling API developed by Jonathan Anderson, which facilitates the implementation of Real-Time Java scheduler objects that work with the RTJVM to provide application level scheduling, similar to the Metascheduler discussed in section 8.1. The RTSJ-Metascheduler will be described in detail in a forthcoming document.

8.2.1 Single Node Experiments

In order to compare the results of the scheduling algorithms HUA and AUA we implemented DASA and a simplified variant of HUA called HUA-Non-Preemptive (or HUA-NP). DASA does not consider handlers for scheduling until failures occur. When a thread fails, DASA then considers its handler for scheduling just like a regular thread, resulting in zero NBI. Similar to DASA HUA-NP also does not consider handlers for scheduling until failures occur. However, when a thread fails, unlike DASA HUA-NP immediately runs the thread handler non-preemptively till completion, resulting in a worst-case and best-case NBI of one handler execution time. In this way, HUA-NP seeks to accrue as much utility as possible by excluding handlers from schedule construction (and thus is more greedy than HUA), while maintaining an upper bound on handler completion. Thus, DASA and HUA-NP are good candidates for a comparative study as they represent two interesting end points of the NBI-versus-handler-completion-time tradeoff space.

Our test application created several periodic threads that consume a certain amount of processor time, request a shared resource, and periodically check for abort exceptions. Each

thread created had a unique execution time, period, and maximum utility. These parameters were assigned based on three PUD-based thread classes that were used: *high*, *medium*, and *low*. The classes differed in thread execution times, thread periods, and threads PUDs by one order of magnitude. The classes, however, differed in handler execution times, handler periods, and handler PUDs only by a small factor. Within each class, thread execution times and thread PUDs were higher than that of their handler execution times and handler PUDs, respectively, by one order of magnitude. For all the experiments, an even number of threads from each of the three classes were used. Thus, the three classes give the algorithms a rich mixture of thread properties to exhibit their NBI and handler completion behaviors.

Our metrics to evaluate AUA and HUA included the NBI, Handler Completion Time (HCT), Accrued Utility Ratio (AUR), and Deadline Miss Ratio (DMR). HCT is the duration between a handler's completion time and it's release time. AUR is the ratio of the total accrued utility to the maximum possible total utility (possible if every released thread completes before its termination time). DMR is the ratio of the number of threads that missed their termination times to the number of released threads.

We manipulated five variables during our experiments: (1) the percentage of failed threads, (2) system load caused by normal tasks, (3) system load caused by handlers, (4) the ratio of handler execution time to normal task execution time, and (5) the number of shared resources within the system. The variables affect the system's "stress factor" and influence the four metrics.

We measured the four metrics under a constant value for these variables, except for the failure percentage, which was varied between 0% and 95%. To vary the failure percentage, the set of threads that must fail for a given percentage must be repeatable. However, to have a repeatable set of discrete failures (i.e., not a random distribution), the actual percentage of failures may be slightly off from the predicted value—e.g., if an experiment had 50 threads

and 25% of them needed to be failed, it is impossible to fail 12.5 threads; thus the failure percentage would be 24% or 26%.

Normal task load was 150%, handler load was 90%, and the ratio of handler execution to normal execution was 50%. We first focused on zero shared resources and then considered shared resources.

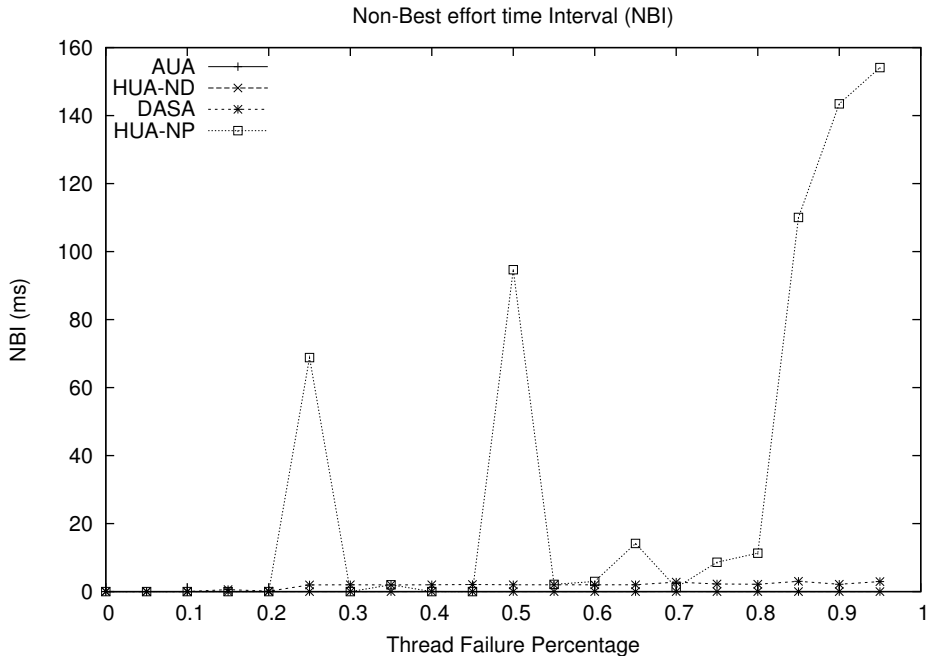

Figure 8.6: Non-Best effort time Interval (NBI)

Figure 8.6 shows the measured NBI of DASA, HUA-NP, AUA, and HUAunder increasing number of failures. We observe that HUA and AUA provide smaller NBI measurements than DASA and HUA-NP.̇They have lower NBI than DASA because DASA is unlikely to execute low-PUD threads like handlers. Thus, it is likely to keep them pending and incur a non-zero NBI due to scheduler overhead when a high PUD thread arrives. HUA and AUA have smaller NBI measurements than HUA-NP because HUA-NP will always have a non-zero NBI when a high PUD thread arrives during its non-preemptive handler execution. Note, the only time HUA will have a non-zero NBI is when a high PUD thread arrives with such little slack that the

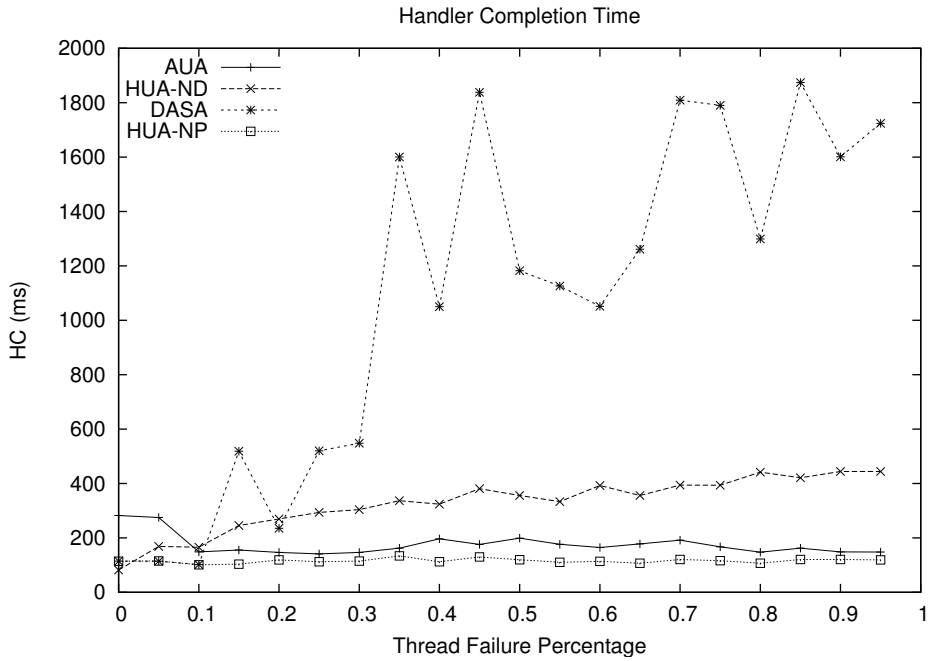

Figure 8.7: Handler Completion Time

pending handlers cannot fit within that slack. AUA will have a non-zero NBI whenever an arriving thread's handler is infeasible with all currently accepted thread handlers or whenever an LCT has been reached.

Figure 8.7 shows the average HCTs for AUA HUA DASA, and HUA-NPİn general, DASA's HCTs are highest and rather inconsistent, HUA-NP's are smallest and very consistent, AUAś are just a little larger than HUA-NP's, and HUA's are somewhat consistent, but always within a certain bound. As DASA was not designed to bound HCTs, it makes sense that its HCTs would be larger than the other two algorithms. Likewise, it makes sense that HUA-NP would have the least average HCT as the handler is run to completion when it is released. To allow more threads to be scheduled, AUA and HUA do not immediately run the handler when it is released. This delay in running the handler causes HUA's and AUA's average HCT to be higher than HUA-NP's. However, as both AUA and HUA were designed to provide a finite bound on HCT, they will generally be smaller than DASA's.

Figures 8.6 and 8.7 also indicate that the trends acquired from our experiments display a less than smooth response to changes in failure percentage. (Figures 8.8 and 8.9 also display this behavior.) This is likely due to the way the failure percentage is varied. Since the set of threads that fail for a given failure percentage is not a strict subset of the set of threads that fail for a larger failure percentage, it is possible that lower-PUD threads may fail at higher failure percentages. Thus, algorithms like DASA and HUA-NP may find a more beneficial schedule at higher failure percentages.

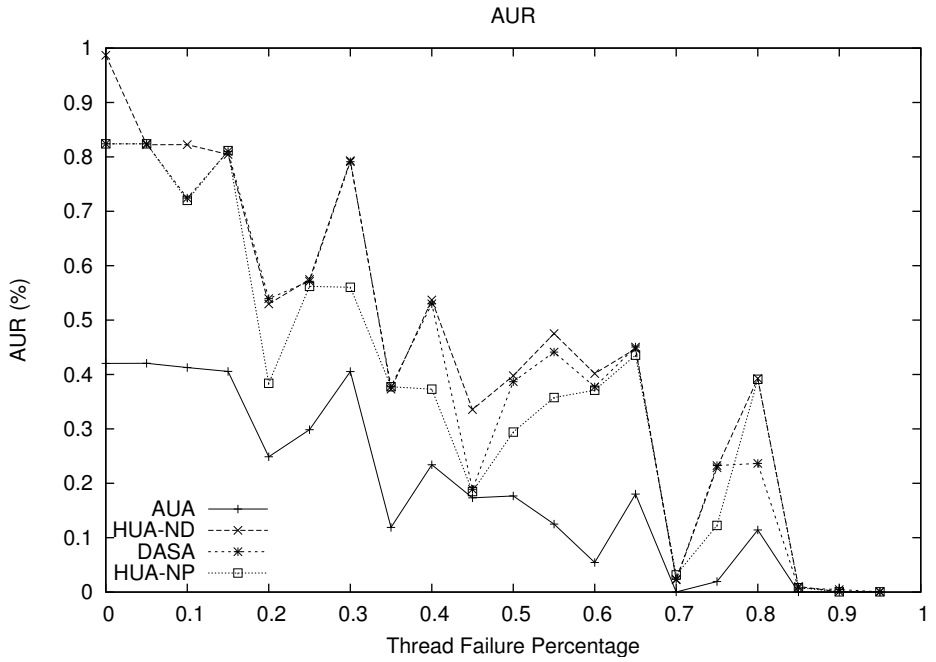

Figure 8.8: Accrued Utility Ratio

Figure 8.8 shows how the AUR of each algorithm is affected under increasing failures. In general, HUA will have a lower AUR as it reserves a portion of its schedule for handlers which generally have lower PUDs or may not even execute. However, as can be seen from the figure, HUA has an AUR that is comparable to, if not better than that of DASA and HUA-NP for this thread set. This is because DASA only analyzes handlers that have been released. This limits DASA's ability to discern whether it would be more beneficial to abort

the thread and run its handler instead. As HUA has no such limitation, it can better decide whether to run the thread or abort the thread and run its handler. AUA has the lowest AUR and this is likely due to its limited view of what a handler's termination time should be, namely the termination time of the thread.

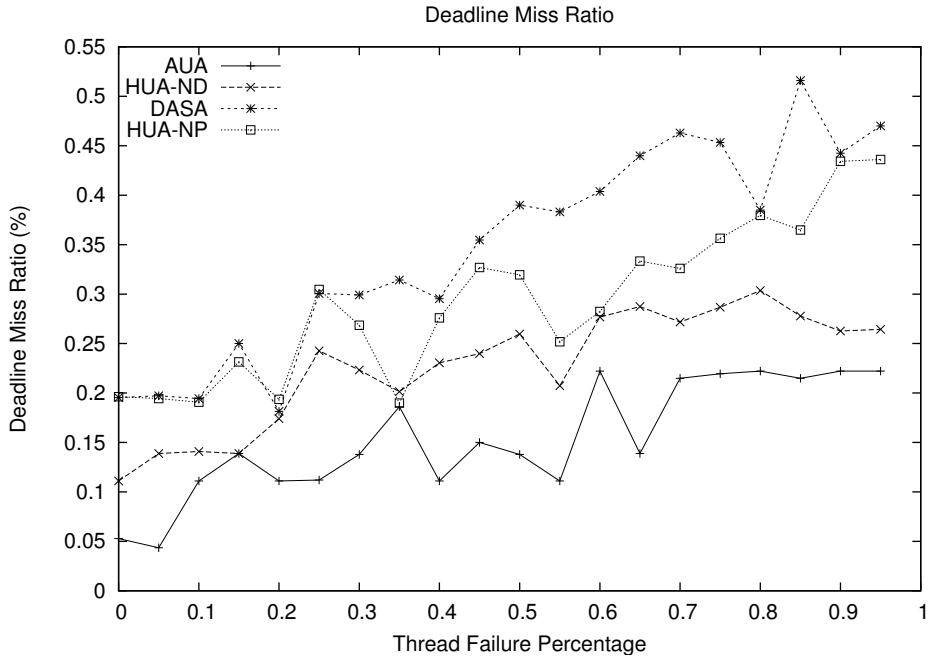

Figure 8.9: Deadline Miss Ratio

Figure 8.9 displays the measured DMR under increasing failures. As the number of failures increases, the number of termination times (or deadlines) missed also increases. This is due to the added load that handlers put on the system. In the case of DASA, this load is completely unforeseen and as the handlers have less PUD than most normal threads, DASA may never schedule them causing their termination times to be missed. Thus, DASA is affected most by increased failures. While the handler load is also unanticipated for HUA-NP, the effects are mitigated somewhat due to HUA-NP's non-preemptive handler execution property. Because HUA takes the handler load into consideration when forming a schedule, the extra load on the system affects HUA is smaller than either DASA or HUA-NP˙AUA has the smallest DMR

because it was designed to minimize the number of misses.

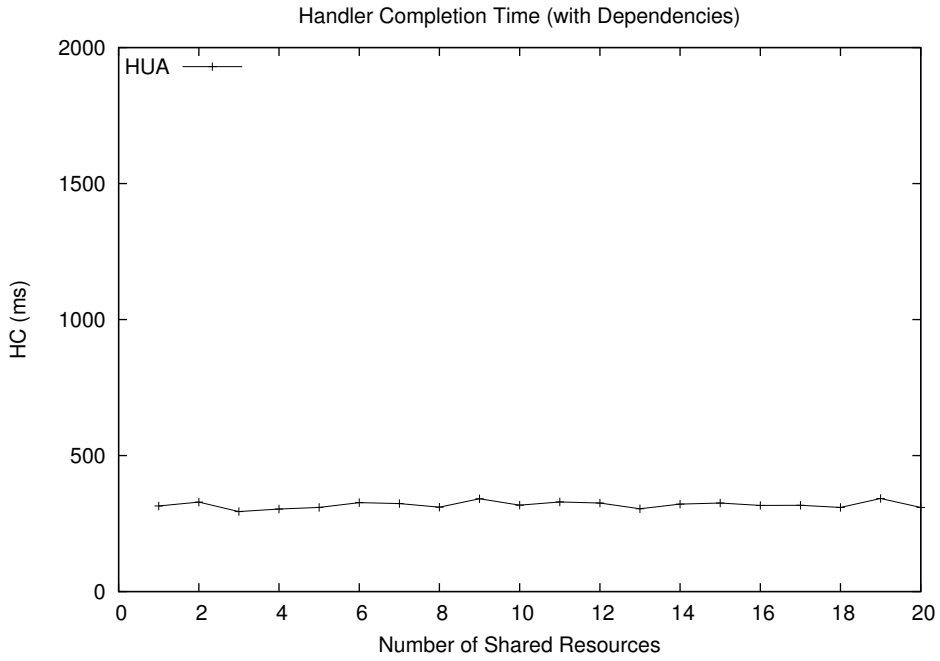

Figure 8.10: Handler Completion Time with Dependencies

Figure 8.10 and Figure 8.11 show the average HCT and average NBI of HUA under increasing number of shared resources. From the figures, we observe that HUA's HCT and NBI are unaffected by dependencies that arise between threads due to shared resources. The AUA plots look similar and are not included in order to avoid redundancy.

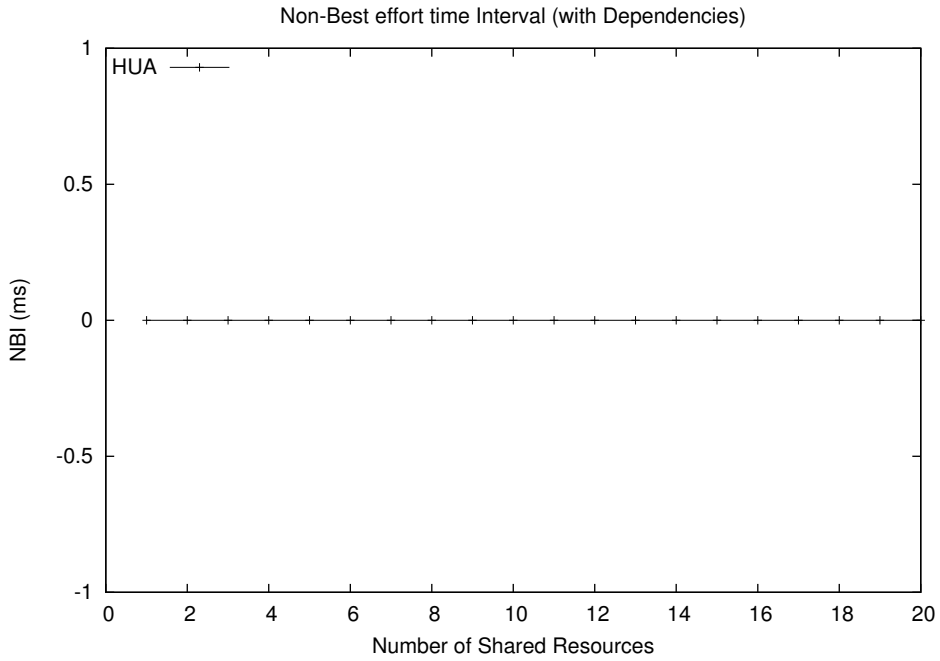

Figure 8.11: Non-Best effort time Interval (NBI) with Dependencies

8.2.2 Multi-Node Experiments

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.

TPR: 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.

For the purposes of evaluating the TPR protocol, we were only concerned with simulating node failures. Such failures were initiated when the master node sent a FAIL command to a

slave node in the system. The slave that received the FAIL command would then proceed to ignore all incoming communication and disable all outgoing communication. For all intents and purposes, the failed node would no longer be in the system.

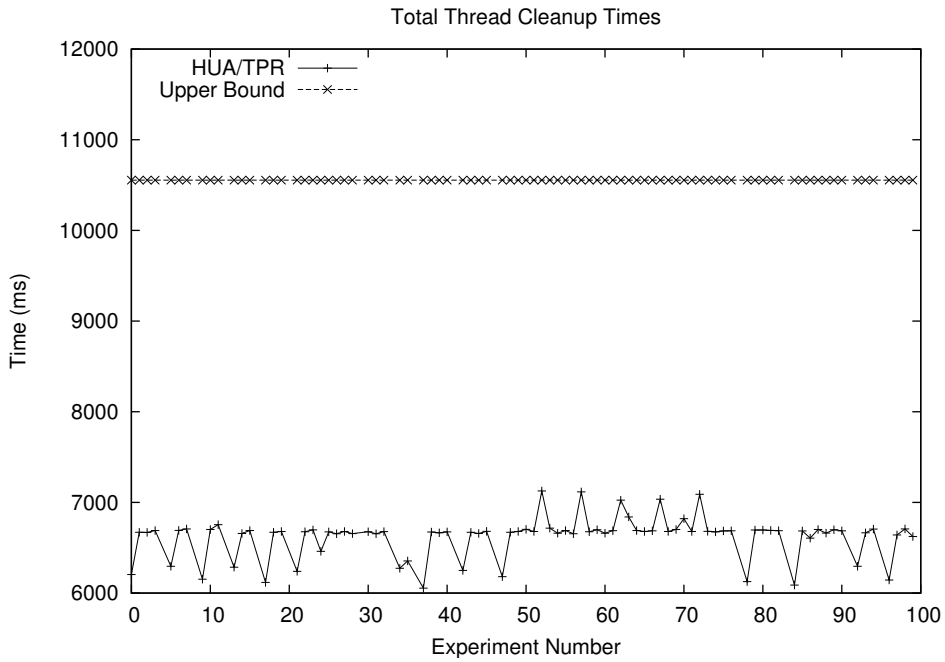

Figure 8.12: Total Thread Cleanup Times

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 8.12 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/TPRsatisfies its cleanup upper bound, validating Theorem 12.

In order for the Total Thread Cleanup Time to satisfy the HUA/TPR cleanup bound, TPR must detect a failure within a certain amount of time as determined by the protocol parameters (e.g., polling interval t_p). Figure 8.13 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

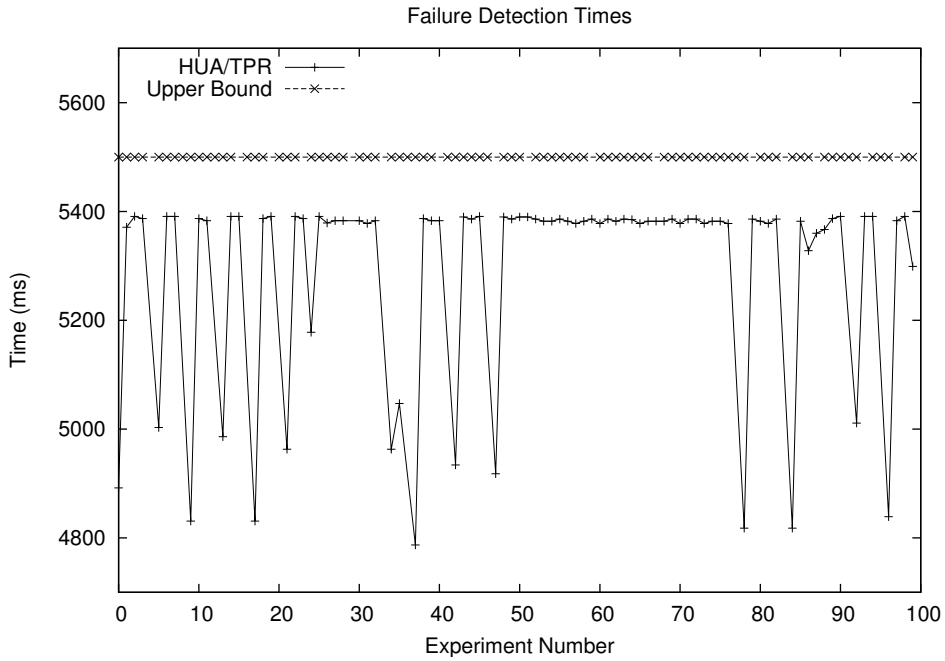

Figure 8.13: Failure Detection Times

bound on failure detection.

The variation observed in Figure 8.13 is actually less than the theoretic variation of t_p (1 second for these experiments) as described in Lemma 9. This variation also occurs in Figure 8.14 for the same reason.

For a thread to recover from a thread break, a new head must be established and orphans must be notified to clean themselves up. Therefore, the last time that must be bounded in order for HUA/TPR to achieve an upper limit on orphan cleanup is the time it takes for the protocol to determine and notify a thread of its new head. We measure this as the New-Head Notification Time. Figure 8.14 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 8.15 shows the thread completion times of experiments 1) with failures and TPR,

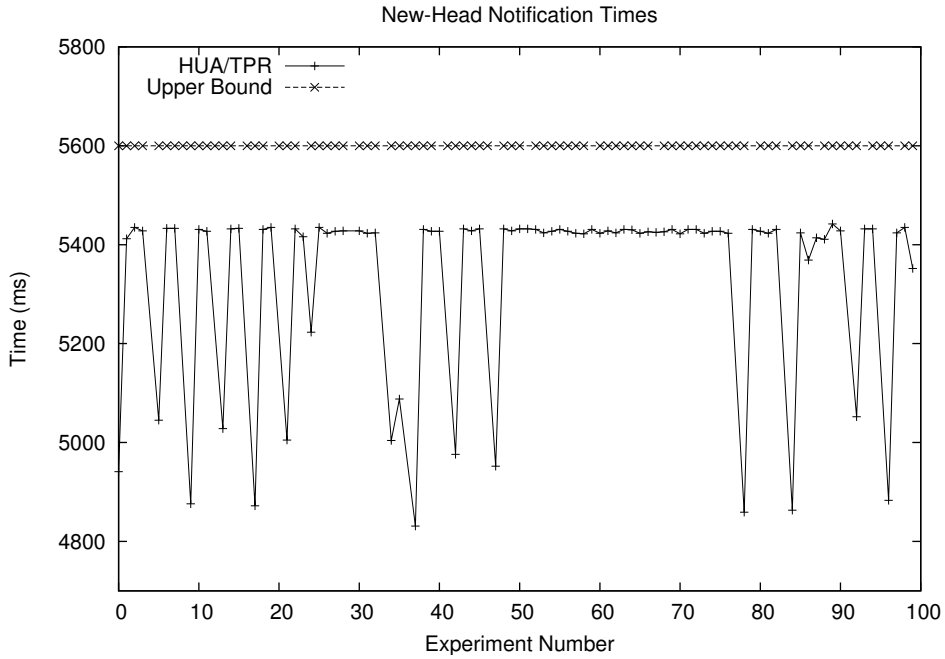

Figure 8.14: New-Head Notification Times

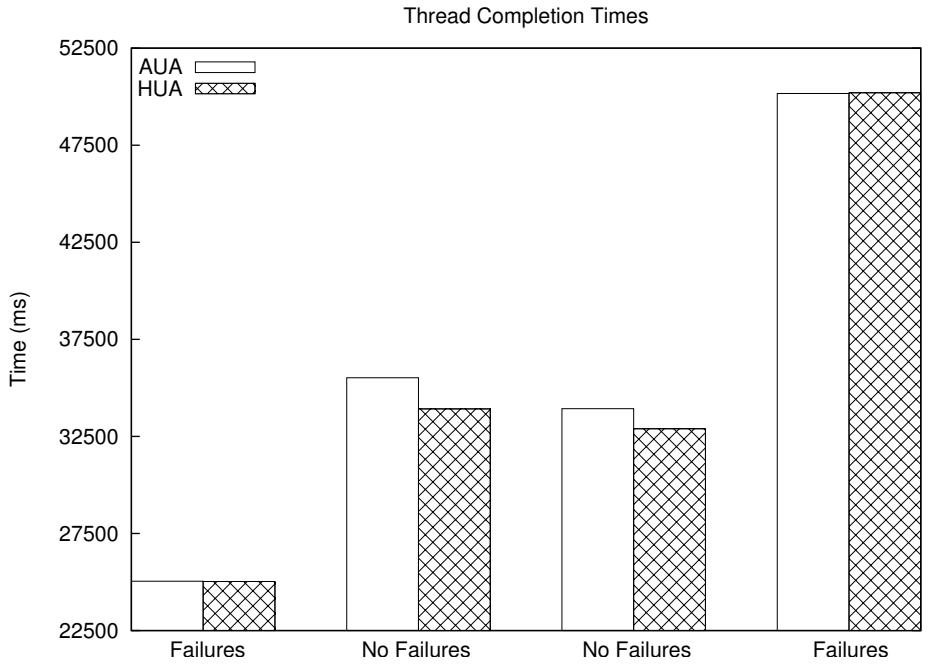

Figure 8.15: Thread Completion Times

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

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 8.15 as the experiments with failures and with TPR had the shortest completion times.

One of the most interesting aspects of Figure 8.15 is the contrast between the experiments without failures. This contrast shows the overhead that TPR incurs when there are no failures present. Another interesting aspect of the figure is the large completion times for experiments with failures, but without TPR. The DRTSJ platform that we used for implementing HUA/TPR enforces a simple, tunable failure detection scheme in the absence of a thread integrity protocol. We purposely chose a large failure detection delay to convey the idea that the threads would never complete without any kind of failure detection and are subject to longer than necessary completion times if the detection scheme is naive.

Figure 8.6 shows the NBI of AUA and HUA under increasing number of thread failures. We observe that AUA has a higher NBI than HUA validating Theorems 5 and 6. This difference in NBI is due to AUA's admission control policy [8]: AUArejects 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.

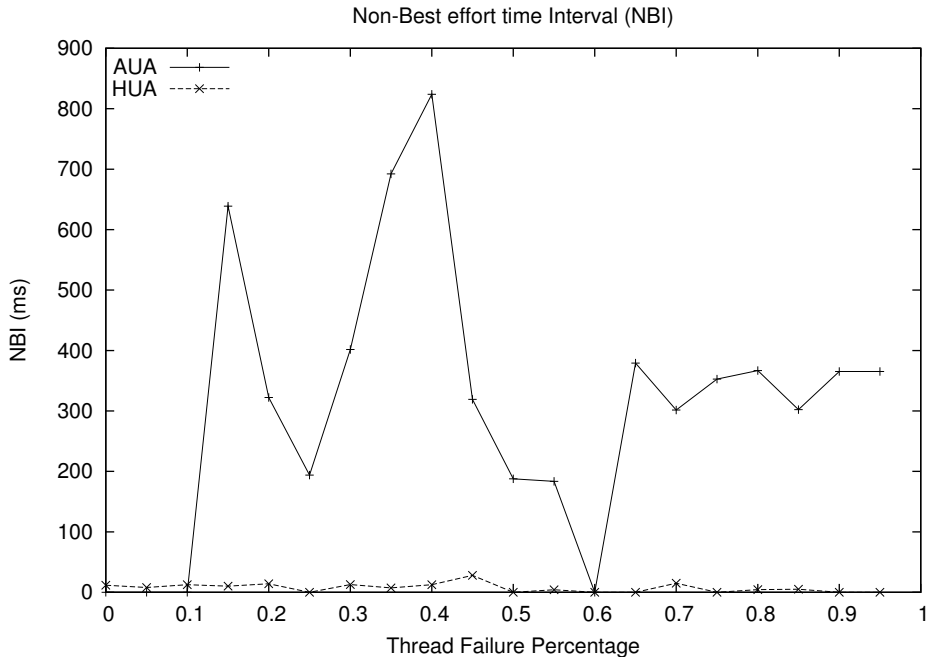

Figure 8.16: Non-Best-effort time Interval (NBI)

The variation of the NBI observed in Figure 8.16 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.

D-TPR and W-TPR: All figures presented below show results collected for tests scheduled using HUA. The differences between AUA and HUA are well documented so, to avoid redundancy, tests using AUA have not been included here.

Figures 8.17 and 8.18 show the measured cleanup times for HUA/D-TPR and HUA/W-TPR, respectively. The cleanup times are plotted against the protocols' cleanup upper bound times for the thread set used in our experiments. From the figures, we observe that both HUA/D-TPR and HUA/W-TPR satisfy their cleanup upper bound, thereby validating Theorems 19 and 23.

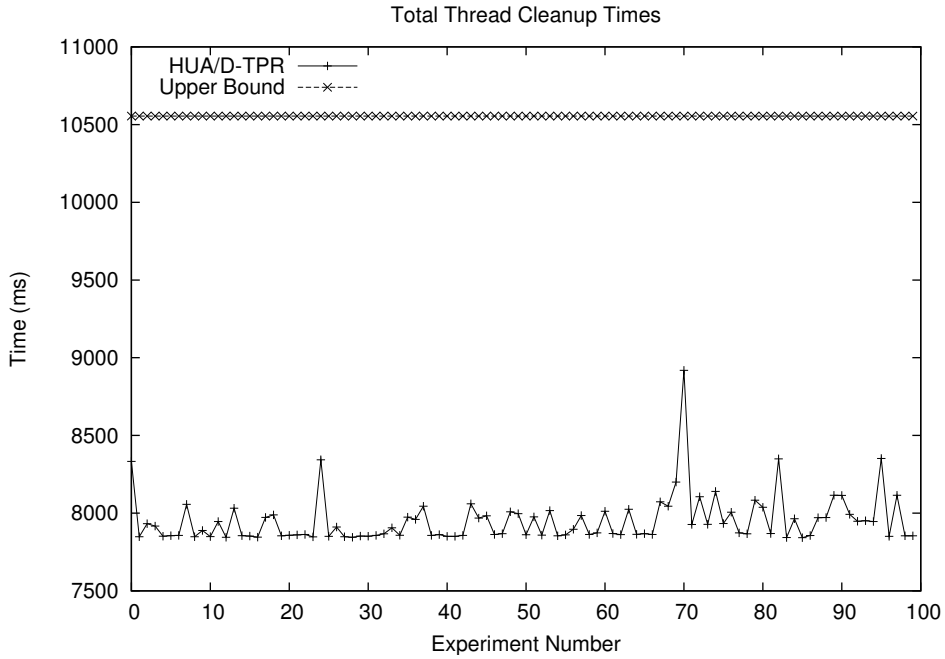

Figure 8.17: D-TPR Thread Cleanup Times

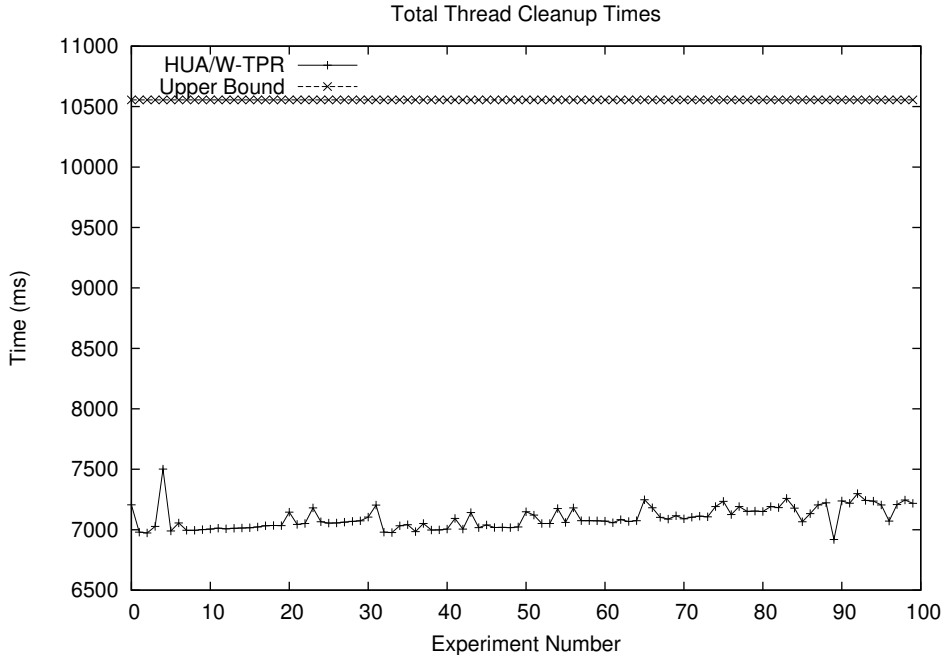

Figure 8.18: W-TPR Thread Cleanup Times

Figures 8.19 and 8.20 show the thread completion times of experiments 1) with failures and D-TPR/W-TPR, 2) without failures but with D-TPR/W-TPR, 3) without failures and without D-TPR/W-TPR, and 4) with failures but without D-TPR/W-TPR. By measuring the thread completion times under these scenarios, we measure the overhead each protocol incurs in terms of the increase in thread completion times.

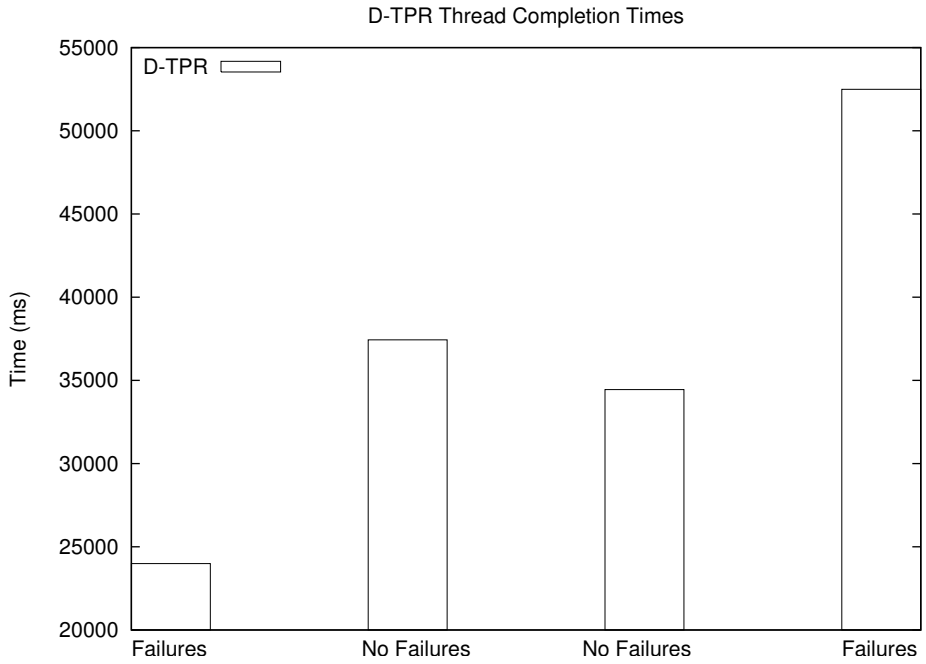

Figure 8.19: D-TPR Thread Completion Times

Figure 8.19 shows the completion times for experiments with and without D-TPR. We observe that the completion times of successful threads without D-TPR is smaller than that with D-TPR. This is to be expected as D-TPR incurs a non-zero overhead. However, we also observe that the completion times of failed threads with D-TPR are shorter than even the completion times of successful threads without D-TPR. This is because, orphan cleanup can occur in parallel with the continuation of a repaired thread, allowing the repaired thread to finish without waiting for all orphans to run to completion. A successful thread, on the other hand, must wait for all sections to finish before it can complete, increasing its completion

time. Figure 8.19 also shows that failed threads with D-TPR complete much more quickly than failed threads with no D-TPR support.

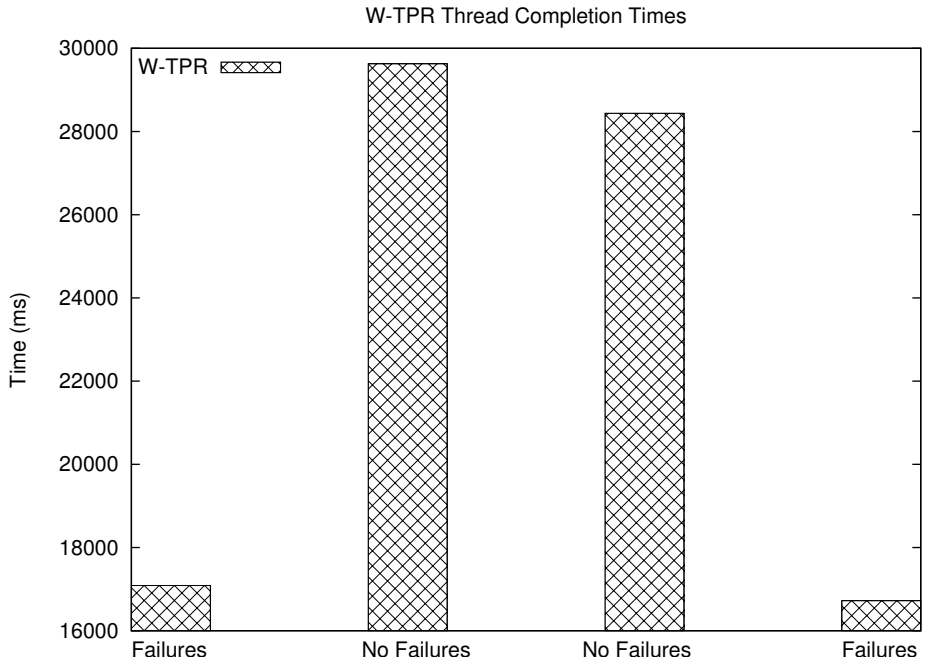

Figure 8.20: W-TPR Thread Completion Times

Figure 8.20 shows completion times for experiments run with and without W-TPR. As the figure shows, the measurements taken in the absence of W-TPR are only slightly lower than the measurements taken in the presence of W-TPR. We observe that W-TPR incurs relatively little overhead while providing the properties discussed in Chapter 7.

Chapter 9

Conclusions and Future Work

We have presented two real-time scheduling algorithm called AUA and HUA and we have presented three distributable thread integrity protocols called TPR, D-TPR, and W-TPR. Together, the algorithms and the protocols schedule and provide thread integrity for threads across a system in the Real-Time CORBA Case II model. In addition, we provide bounds on the worst-case fault detection and cleanup time for threads experiencing partial failures.

The experimental results presented demonstrate the effectiveness of the scheduling algorithms scheduling a variety of task loads induced by threads created with the DRTSJ distributable thread framework. Furthermore, we argue that this suite provides a useful framework for implementing resilient distributed computational activities in systems subject to partial (crash) failures.

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

for ensuring deterministic safety while providing graceful degradation in overloads.

This work can be extended in several directions. Examples include relaxing the upper bounds on communication delays for TPR and D-TPR allowing DTs to share non-CPU resources under mutual exclusion constraints, allowing DTs to have nested time constrained scopes, and considering abort handling models where DTs are considered for execution irrespective of the feasibility of the abort handlers.

Bibliography

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

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

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

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

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

[6] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the Alpha real-time distributed kernel. In *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, April 1992.

[7] T. Clausen, P. J. (editors), C. Adjih, A. Laouiti, P. Minet, P. Muhlethaler, A. Qayyum, and L.Viennot. Optimized link state routing protocol (OLSR). RFC 3626, October 2003. Network Working Group.

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

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

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

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

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

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

[14] J. Goldberg, I. Greenberg, R. K. Clark, E. D. Jensen, K. Kim, and D. M. Wells. Adaptive fault-resistant systems (chapter 5: Adpative distributed thread integrity). Technical Report csl-95-02, Computer Science Laboratory, SRI International, Menlo Park, CA., January 1995. http://www.csl.sri.com/papers/sri-csl-95-02/.

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

[16] E. D. Jensen and J. D. Northcutt. Alpha: A non-proprietary operating system for large, complex, distributed real-time systems. In *IEEE Workshop on Experimental Distributed Systems*, pages 35–41, 1990.

[17] E. D. Jensen, A. Wellings, R. Clark, and D. Wells. The distributed real-time specification for Java: A status report. In *Proceedings of The Embedded Systems Conference*, 2002.

[18] D. Johnson, D. Maltz, and J. Broch. *DSR The Dynamic Source Routing Protocol for Multihop Wireless Ad Hoc Networks*.

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

[20] G. Koren and D. Shasha. D-over: An optimal on-line scheduling algorithm for overloaded real-time systems. In *IEEE Real-Time Systems Symposium*, pages 290–299, December 1992.

[21] P. Li, B. Ravindran, H. Cho, and E. D. Jensen. Scheduling distributable real-time threads in Tempus middleware. In *IEEE Conference on Parallel and Distributed Systems*, pages 187 – 194, July 2004.

[22] P. Li, B. Ravindran, et al. A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems. *IEEE Trans. Software Engineering*, 30(9):613 – 629, Sept. 2004.

[23] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

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

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

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

[27] J. D. Northcutt and R. K. Clark. The Alpha operating system: Programming model. Archons Project Technical Report 88021, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1988.

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

[29] K. Romer. Time synchronization in ad hoc networks, 2001.

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