Exploiting Slack for Scheduling Dependent, Distributable Real-Time Threads in Mobile Ad Hoc Networks

Kai Han*, Binoy Ravindran*, and E. D. Jensen‡

*ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{khan05,binoy}@vt.edu

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

Abstract

We consider scheduling distributable real-time threads with dependencies (e.g., due to synchronization) in mobile ad hoc networks, in the presence of node/link failures, message losses, and dynamic node joins and departures. We present a distributed real-time scheduling algorithm called RTG-DS. The algorithm uses a gossip-style protocol for discovering eligible nodes, node/link failures, and message losses. In scheduling local thread sections, it exploits thread slacks to optimize the time available for gossiping. We prove that RTG-DS probabilistically bounds distributed blocking times and distributed deadlock detection and notification times. Thereby, it probabilistically satisfies end-to-end thread time constraints. We also prove that RTG-DS probabilistically bounds failure-exception notification times for failed threads (so that their partially executed sections can be aborted). Our simulation results validate RTG-DS's effectiveness.

1. Introduction

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

An abstraction for programming causal, multi-node sequential behaviors and for enforcing end-to-end properties on them is *distributable threads* [2,17]. They first appeared in the Alpha OS [17], and now constitute the first-class programming and scheduling abstraction for multi-node sequential behaviors in Sun's emerging Distributed Real-Time Specification for Java [2]. In the rest of the paper, we will refer to distributable threads as *threads*, unless qualified.

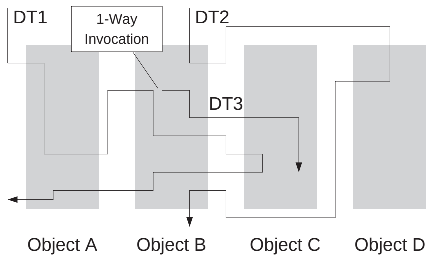

Figure 1. Distributable Threads

A thread is a single logically distinct (i.e., having a globally unique ID) locus of control flow movement that extends and retracts through local and (potentially) remote objects. A thread carries its execution context as it transits node boundaries, including its scheduling parameters (e.g., time constraints, execution time), identity, and security credentials. The propagated thread context is intended to be used by node schedulers for resolving all node-local resource contention among threads such as that for node's physical and logical resources (e.g., CPU, I/O, locks), according to a discipline that provides acceptably optimal system-wide timeliness. Thus, threads constitute the abstraction for concurrency and scheduling. Fig. 1 shows the execution of three threads [18].

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

We consider threads as the programming and scheduling abstraction in *ad hoc networks* (e.g., those without a fixed network infrastructure, including mobile and wireless networks [3]), in the presence of application- and network-induced uncertainties. The uncertainties include resource overloads (due to context-dependent thread execution times), arbitrary thread arrivals, arbitrary node failures, and transient and permanent link failures (causing varying packet drop rate behaviors). Another distinguishing feature of motivating applications for this model (e.g., [7]) is their relatively long thread execution time magnitudes—e.g., milliseconds to minutes. Despite the uncertainties, such applications desire strong assurances on end-to-end thread timeliness behavior. Probabilistic timing assurances are often appropriate.

When threads mutually-exclusively share non-CPU resources (e.g., disks, NICs) at a node using lock-based synchronizers, distributed dependencies can arise, causing distributed blockings and deadlocks. For example, a thread A may lock a resource on a node and may make a remote invocation, carrying the lock with it. Thread B may later request the same lock and will be blocked, until A unwinds back from its remote invocation and releases the lock. Unbounded blocking time can degrade system-wide timeliness optimality—e.g., B may have a greater urgency than A. Further, distributed deadlocks can occur when threads A and B block on each other for remotely held locks. Unbounded deadlock detection and resolution times can also degrade timeliness optimality.

When a thread encounters a node/link failure, partially executed thread sections may be blocked on nodes that are upstream and downstream of the failure point, waiting for the thread to unwind back from invocations that are further downstream to them. Such sections must be notified of the thread failure, so that they can respond with application-specific exception handling actions—e.g., releasing handlers for execution that abort the sections, after releasing and rolling-back resources held by them to safe states (under a termination model). Untimely failure notifications can degrade timeliness optimality—e.g., threads unaffected by a partial failure may become indefinitely blocked by sections of failed threads.

In this paper, we present an algorithm called *Real-Time Gossip algorithm for Dependent threads with Slack scheduling* (or RTG-DS) that provides assurances on thread time constraint satisfactions in the presence of distributed dependencies in ad hoc networks. At its core, RTG-DS is a gossip protocol (e.g., [13] and references therein). The algorithm uses gossip-style communication for propagating thread scheduling parameters, and for discovering nodes (hosting thread sections), and node/link failures. Further, the algorithm schedules thread sections by exploiting thread slack in a way that enhances the time available for gossiping.

We prove that thread blocking times and deadlock detection and notification times are probabilistically bounded under RTG-DS. Consequently, we prove that thread time constraint satisfactions' are probabilistically bounded. We also prove that RTG-DS probabilistically bounds failure-exception notification times for partially executed sections of failed threads. Our simulation studies verify the algorithm's effectiveness.

End-to-end real-time scheduling/resource management has been previously studied (e.g., [1, 4, 5, 17, 24, 25]), but these are limited to fixed infrastructure networks. Real-time assurances in ad hoc networks has been studied (e.g., [10, 15, 26]), but these exclude dependencies, which is precisely what RTG-DS targets.

Our work builds upon our prior work in [9] that presents the RTG-D algorithm. While RTG-DS uses a slack-based thread scheduling approach, RTG-D uses the Dependent Activity Scheduling Algorithm (DASA) in [8] for thread scheduling. We compare RTG-DS with RTG-D in this paper and illustrate RTG-DS's superiority. Further, RTG-D does not consider deadlock detection and notification, and failure-exception notification, while RTG-DS provides probabilistic assurances on such notification times. Thus, the paper's contribution is the RTG-DS that provides probabilistic end-to-end timing assurances (time constraint satisfactions and failure recovery times) in the presence of distributed blockings and deadlocks.

The rest of the paper is organized as follows: In Section 2, we discuss the models of RTG-DS (these are the same as that of RTG-D) and state the algorithm objectives. Section 3 presents RTG-DS. We analyze RTG-DS in Section 4. In Section 5, we report our simulation studies. We conclude the paper and identify future work in Section 6.

2. Models and Algorithm Objectives

2.1. Task Model: 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*. Thus, a thread can be viewed as being composed of a concatenation of thread segments.

A thread's initial segment is called its *root* and its most recent segment is called its *head*. The head of a thread is the only segment that is active. A thread can also be viewed as being composed of a sequence of *sections*, where a section is a maximal length sequence of contiguous thread segments on a node. A section's first segment results from an invocation from another node, and its last segment performs a remote invocation. More details on threads can be found in [2,17,18].

Execution time estimates of the sections of a thread are assumed to be known when the thread arrives at the respective nodes. Note that a section's execution time estimate is that of the contiguous set of thread segments that starts from the first thread segment executed on the node and ends with the first remote invocation made from the node. The time estimate includes that of the section's normal code and its exception handler code, and can be violated at run-time (e.g., due to context dependence, causing overloads).

Each object transited by threads is uniquely hosted by a node. Threads may be created at arbitrary times at a node. Upon creation, the number of objects (and the object IDs) on which they will make subsequent invocations are known. The identifier of the nodes hosting the objects, however, are unknown at thread creation time, as nodes may dynamically fail, or join, or leave the system. Thus, eligible nodes have to be dynamically discovered as thread execution progresses.

The sequence of remote invocations and returns made by a thread can be estimated by analyzing the thread code (e.g., [16]). The maximum number of sections of a thread is thus assumed to be known.

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

2.2. Timeliness Model

Each thread's time constraint is specified using a time/utility function (or TUF) [11]. A TUF specifies the utility of completing a thread as a function of that thread's completion time. Fig. 2 shows three example downward "step" shaped TUFs. A thread's TUF decouples its *importance* and *urgency*—i.e., urgency is measured as a deadline on the X-axis, and importance is denoted by utility on the Y-axis. This decoupling is significant, as a thread's urgency is sometimes orthogonal to its relative importance—e.g., the most urgent thread is the least important, and vice versa; the most urgent is the most important, and vice versa.

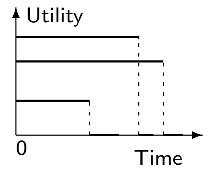

Figure 2. Step TUFs

A thread T_i's TUF is denoted as $U_i(t)$. Classical deadline is unit-valued—i.e., $U_i(t) = \{0, 1\}$, since importance is not considered. Downward step TUFs generalize classical deadlines where $U_i(t) = \{0, \{n\}\}$. We focus on downward step TUFs, 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 termination time X_i, which, for a downward step TUF, is its discontinuity point. Further, we assume that $U_i(t) > 0, \forall t \in [I_i, X_i]$ and $U_i(t) = 0, \forall t \notin [I_i, X_i], \forall i$.

When thread time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued thread utility—e.g., maximizing the sum of the threads' attained utilities. Such criteria are called *utility accrual* (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that consider UA criteria are called UA sequencing algorithms. The criteria may also include other factors such as resource dependencies. Several UA algorithms such as DASA are presented in the literature [20]. We derive RTG-DS's local thread section scheduling algorithm from DASA, and compare it with DASA.

2.3. Exceptions and Abortion Model

If a thread has not completed by its termination time, a failure-exception is raised, and exception handlers are immediately released and executed for aborting all partially executed thread sections. The handlers are assumed to perform the necessary compensations to avoid inconsistencies (e.g., rolling back resources held by the sections to safe states) and other actions that are required for the safety and stability of the external state. This abortion model is similar to that in transactional paradigms (e.g., [23]).

Note that, once a thread fails to meet its termination time, the scheduler at the node where the thread's termination time expires will immediately raise the failure exception. At all other (upstream) nodes, where the thread has partially executed and is waiting for the head to unwind back from invocations, a notification for the exception must be delivered. Delivery of that notification is done by RTG-DS.

We consider a similar abortion model for thread failures, for resolving deadlocks, and for resolving thread blocks when a blocking thread is aborted to obtain greater utility (similar to transactional abortions [23]). The scheduler at the node where these situations are detected will raise the failure exception. At all other (upstream) nodes, where the thread has partially executed, RTG-DS delivers the failure exception.

2.4. Resource Model

Thread sections can access non-CPU resources (e.g., disks, NICs) located at their nodes during their execution, which are serially reusable. Similar to fixed-priority resource access protocols [22] and that for TUF algorithms [8, 14], we consider a single-unit resource model. Resources can be shared under mutual exclusion constraints. A thread may request multiple shared resources during its lifetime. The requested time intervals for holding resources may be nested, overlapped or disjoint. Threads explicitly release all granted resources before the end of their executions.

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

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

2.5. System Model

The system consists of a set of processing components, generically referred to as *nodes*, denoted $N = \{n_1, n_2, n_3, \ldots\}$, communicating through bidirectional wireless links—e.g., [3]. A basic unicast routing protocol such as DSR [12] is assumed for packet transmission. 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 [21]. Nodes may dynamically join or leave the network. We assume that the network communication delay follows some non-negative probability distribution—e.g., the Gamma distribution. Nodes may fail by crashing, links may fail transiently or permanently, and messages may be lost, all arbitrarily.

2.6. Objectives

Our goal is to design an algorithm that can schedule threads with probabilistic termination-time satisfactions in the presence of (distributed) blockings and deadlocks—i.e., establish probabilistically-satisfied blocking time and deadlock detection and notification time for a thread, so that the probability of the thread for satisfying its termination time can be computed. We also desire to maximize the total thread accrued utility. Moreover, the time needed to notify partially executed sections of a failed thread (so that handlers for aborting thread sections can be released) must also be probabilistically bounded.

Note that maximizing the total utility subsumes meeting all termination times as a special case. When all termination times are met (during underloads), the total accrued utility is the optimum possible. During overloads, the goal is to maximize the total utility as much as possible, thereby completing as many important threads as possible, irrespective of their urgency.

3. The RTG-DS Algorithm

RTG-DS builds upon RTG [10]. Some aspects of RTG-DS (e.g., determining a thread's next destination node) are the same as those of RTG. In describing RTG-DS, we describe the entire algorithm for completeness, but summarize parts that are the same as that of RTG for brevity. We first overview RTG-DS.

When a thread arrives at a node, RTG-DS decomposes the thread's end-to-end TUF into a set of local TUFs, one for each of the sections of the thread. The decomposition is done using the thread's scheduling parameters including its end-to-end TUF, number of sections, section execution time estimates that the thread presents to RTG-DS upon arrival. Local TUFs are used for thread scheduling on nodes.

When a thread completes its execution on a node, RTG-DS must determine the thread's next destination node. In order to be robust against node/link failures, message looses, and node joins/departures, RTG-DS uses a gossip-style protocol (e.g., [6]). The algorithm starts a series of synchronous gossip rounds. During each round, the node randomly selects a set of neighbor nodes and queries whether they can execute the thread's next section (as part of the thread's next invocation or return from its current invocation). The number of gossip rounds, their durations, and the number of neighbor nodes (i.e., the "fan-out") are derived from the local TUF's slack, as they directly affect the communication time incurred by gossip, and thereby affect following sections' available local slack.

When a node receives a gossip message, it checks whether it hosts the requested section, and can complete it satisfying its local TUF (propagated with the gossip message). If so, it replies back to the node where the gossip originated (referred to as the original node). If not, the node starts a series of gossip rounds and sends gossip messages (like the original node).

If the original node receives a reply from a node before the end of its gossip rounds, the thread is allowed to make an invocation on, or return to that node, and thread execution continues. If a reply is not received, the node regards that further thread execution is not possible (due to possible node/link failures or node departures), and releases the section's exception handler for execution. A series of gossip rounds is also immediately started to deliver the failure-exception notification to all upstream sections of the thread, so that handlers may be released on those nodes.

We now discuss each of the key aspects of RTG-DS in the subsections that follow.

3.1. Building Local Scheduling Parameters

RTG-DS decomposes a thread's end-to-end TUF based on the execution time estimates of the thread's sections and the TUF termination time. Let a thread T_i arrive at a node n_j at time t. Let T_i's total execution time of all the thread sections (including the local section on n_j) be Er_i, the total remaining slack time be Sr_i, the number of remaining thread sections (including the local section on n_j) be Nr_i, and the execution time of the local section be $Er_{i,j}$. RTG-DS computes a local slack time $LS_{i,j}$ for T_i as $LS_{i,j} = \frac{Sr_i}{Nr_i - 1}$, if $Nr_i > 1$; $LS_{i,j} = Sr_i$, if $0 \leqslant Nr_i \leqslant 1$.

RTG-DS determines the local slack for a thread in a way that allows the remaining thread sections to have a fair chance to complete their execution, given the current knowledge of section execution-time estimates, in the following way. When the execution of T_i's current section is completed at the node n_j, RTG-DS determines the next node for executing the thread's next section, through a set of gossip rounds. The network communication delay incurred by RTG-DS for the gossip rounds must be limited to at most the local slack time $LS_{i,j}$. The algorithm equally divides the total remaining slack time to give the remaining thread sections a fair chance to complete their execution.

The local slack is used to compute a local termination time for the thread section. The local termination time for a thread T_i is given by $LX_{i,j} = t + Er_{i,j} + LS_{i,j}$. The local termination time is used to test for schedule feasibility, while constructing local section schedules (we discuss this in Section 3.3).

3.2. Determining Next Destination Node

Once the execution of a section completes on a node, RTG-DS determines the node for executing the next section of the thread, through a set of gossip rounds during which the node randomly multicasts with other nodes in the network. RTG-DS determines the number of rounds for "gossiping" (i.e., sending messages to randomly selected nodes during a single gossip round) as follows. Let the execution of T_i's local section on node n_j complete at time t_c. T_i's remaining local slack time is given by $LSr_{i,j} = LX_{i,j} - t_c$.

Note that $LSr_{i,j}$ is not always equal to $LS_{i,j}$, due to the interference that the thread section suffers from other sections on the node. Thus, $LSr_{i,j} \leq LS_{i,j}$. With a gossip period Ψ, RTG-DS determines the number of gossip rounds before $LX_{i,j}$ as $round = LSr_{i,j}/\Psi$. RTG-DS also determines the number of messages that must be sent during each gossip round, called *fan out*, for determining the next node.

RTG-DS divides the system node members into: a) *head* nodes that execute thread sections, and b) *intermediate* nodes that propagate received gossip messages to other members. Detailed procedure-level descriptions of RTG-DS algorithms on head node and intermediate node can be found in [10].

3.3. Scheduling Local Sections

RTG-DS constructs local section schedules with the goals of (a) maximizing the total attained utility from all local sections, (b) maximizing the number of local sections meeting their local termination times, and (c) increasing the likelihood for threads to meet thread termination times, while respecting dependencies.

The algorithm's scheduling events include section arrivals and departures, and lock and unlock requests. When the algorithm is invoked, it first builds the dependency list of each section by following the chain of resource request and ownership. A section i is dependent upon a section j, if i needs a resource which is currently held by j. Dependencies can be local—i.e., the requested lock is locally held, or distributed—i.e., the requested lock is remotely held.

The algorithm then checks for deadlocks, which can be local or distributed (e.g., two threads are blocked on their respective nodes for locks which are remotely held by the other). Deadlocks are detected by the presence of a cycle in the resource graph (a necessary condition). Deadlocks are resolved by aborting that section in the cycle, which will likely contribute the least utility. That section is aborted by executing its handler, which will perform roll-backs/compensations.

Now, the algorithm examines sections in the order of non-increasing *potential utility densities* (or PUDs). Informally, a section's PUD is the total utility accrued by immediately executing it and its dependents divided by the aggregate execution time spent. Thus, a section's PUD measures its "return on investment."

The algorithm inserts each examined section and its dependents into a tentative schedule that is ordered by local slacks, least-slack-first (or LSF). The insertion also respects each section's dependency order.

After insertion, RTG-DS checks the schedule's feasibility with respect to satisfying all inserted sections' local termination times. If infeasible, the inserted section and its dependents are removed. The process is repeated until all sections are examined. Then, RTG-DS selects the least-slack section for execution. If the selected section is remote (because it holds a locally requested lock), the algorithm will speed up it's execution by adding all local dependents' utilities and propagating the aggregate value to it (by gossiping).

We now explain key steps of the algorithm in detail.

3.3.1 Arranging Sections by PUD

The local scheduling algorithm examines sections in non-increasing PUD order to maximize the total accrued utility. Section i's PUD, $PUD_i = \frac{U_i + U(Dep(i))}{c_i + c(Dep(i))}$, where U_i is i's utility, c_i is i's execution time, and $Dep(i)$ is the set of sections on which i is directly or transitively dependent. Note that PUD_i can change over time, since c_i and $Dep(i)$ may change over time.

3.3.2 Determining Schedule Feasibility

RTG-DS determines a node's processor load ρ_R by considering that node's own processor bandwidth, and also by leaving a necessary gossip time interval for each thread section. Let t be the current time, and d_i be the local termination time of section i. ρ_R in time interval $[t,\ d_i]$ is given by:

$$\rho_{R_i}(t) = \frac{\sum_{d_k \leq d_i} c_k(t) + T_{comm}}{(d_i - t)}, \quad T_{comm} \geq LCD$$

where $c_k(t)$ is section k's remaining execution time with $d_k \leq d_i$, and LCD is the lower bound of network communication delay. Different from computing ρ on a single node, RTG-DS adds an additional communication time interval, T_{comm}, to each $c_k(t)$. If a section is the last one of its parent thread, there is no need to consider gossiping time and $T_{comm} = 0$. Without adding T_{comm}, a section may successfully complete, but may not have enough time to find the next destination node. Thus, not only that section's parent thread will be aborted in the end, but also will waste processor bandwidth, which could otherwise be used for other threads' sections.

Suppose there are n sections on a node, and let d_n be the longest local termination time. Then, the total load in $[t, d_n]$ is computed as: $\rho_R(t) = max\rho_{R_i}(t), \forall i$.

3.3.3 Least-Slack Section First (LSF)

RTG-DS selects local sections with the lesser (local) slack time earlier for execution. By doing so, RTG-DS ensures that greater remaining slack time is available for threads to find their next destination nodes.

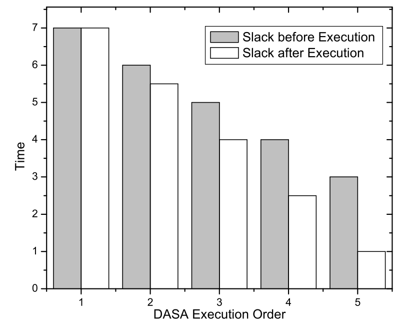

Figure 3. Slack Under DASA

For example, consider five sections with different local slack times. Fig. 3 shows slacks of the sections before and after execution under DASA, on a single node. In the worst case, DASA will schedule sections along the decreasing order of slacks, as shown in Fig. 3. Assuming that the lower bound of network communication time, LCD, is 0.5 time unit, section 5 has only 1 time unit left to gossip (its original local slack time is 3 time units), which makes it more difficult to make a successful invocation on (or return to) another node.

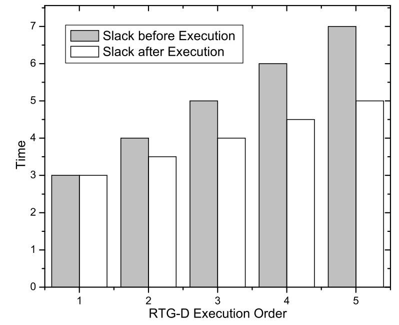

Figure 4. Slack Under RTG-DS

RTG-DS avoids this with the LSF order. In Fig. 4, section 5's remaining local slack remains unchanged after execution, while section 1's slack decreases from 7 to 5 time units, which will slightly decrease its gossip time. Note that RTG-DS gains the same total slack time in these five sections as DASA does, but it allocates slack time more evenly, thereby seeks to give each section an equal chance to complete gossiping.

When checking feasibility, it is important to respect dependencies among sections. For example, section j may depend on section i, thus i must be executed before j to respect the dependency. However, under $LS_{ri} > LS_{rj}$, j will be arranged before i. To resolve this conflict without breaking the LSF order, RTG-DS "tightens" i's local slack time to the same as j's.

RTG-DS's local section scheduling algorithm is described in Algorithm 1.

Algorithm 1: RTG-DS's Local Section Scheduling Algorithm

```
1  Create an empty schedule $\phi$;
2  for each section i in the local ready queue do
3      Compute Dep(i), detecting and resolving deadlocks if any;
4      Compute $PUD_i$;
5  Sort sections in ready queue according to PUDs;
6  for each section i in decreasing PUD order do
7      $\hat{\phi} = \phi$; /* get a copy for tentative changes */
8      if $i \notin \phi$ then
9          CurrentLST = LST(i); /* LST(i) returns the local slack of i */
10         for each PrevS in Dep(i) do
11             if $PrevS \in \phi$ then
12                 if $LST(PrevS) \leq CurrentLST$ then
13                     Continue;
14                 else
15                     LST(PrevS) = CurrentLST;
16                     Remove(PrevS, $\hat{\phi}$, LST); /* Remove PrevS from $\hat{\phi}$ at position LST */
17             Insert(PrevS, $\hat{\phi}$, CurrentLST);
18         if Feasible($\hat{\phi}$) then
19             Insert(i, $\hat{\phi}$, CurrentLST);
20             if Feasible($\hat{\phi}$) then
21                 $\phi = \hat{\phi}$;
22 Select least-slack section from $\phi$ for execution;
```

3.3.4 Utility Propagation

Section i may depend on section j located on the same node or on a different node. For the latter case, RTG-DS propagates i's utility to j in order to speed up j's execution, and thus shorten i's time waiting for blocked resources. The utility is propagated by gossiping to all system members within a limited time interval, as it does in finding the next destination node.

When j's head node receives an utility-propagation message, it has to decide whether to continue executing j, or to immediately abort j and grant the lock to i. This decision is based on Global Utility Density (or GUD), which is defined as the ratio of the owner thread utility to the total remaining thread execution time. Thread PUDs are not used in this case, because this utility comparison involves multiple nodes.

Algorithm 2 describes this decision process. If the decision is to continue j's execution, the node will add i's utility to j's current and previous head nodes, consequently speeding up j's execution (since the scheduler examines sections in the PUD order). If the decision is not to continue j's execution, the node will release j's abort handler, and will start gossiping to 1)

Algorithm 2: RTG-DS's Utility Propagation Algorithm

```
Upon receiving a UP gossip message msg:
COPY(gossip, msg) ;
if $GUD_i > GUD_j$ then
    if $abt_j < er_j$ then
        abort $j$;
        gossip.lsr ← msg.lsr − $abt_j$;
        /* give resource lock to $i$ */
    else
        continue $j$'s execution;
        /* keep resource lock */
        gossip.lsr ← msg.lsr;
else
    gossip.lsr ← msg.lsr;
gossip.round ← gossip.lsr/$\Psi$ ;
gossip.c ← FANOUT(gossip.round);
RTG_GOSSIP(gossip);
```

release j's abort handler's on all previous head nodes of j and 2) grant lock to i. Note that i's utility is only propagated to j's execution nodes after the node from where i requested the lock, because j's other execution nodes do not contribute to this dependency.

3.3.5 Resolving Distributed Deadlocks

Detecting deadlocks between different nodes require all system members to uniformly identify each thread. Thus, when a thread is created, a global ID (or GID) is created for it. With GIDs, it is easier to determine the thread that must be aborted to resolve a distributed deadlock: If $GUD_i > GUD_j$, then i has a higher utility. Then, j is aborted to grant the lock to i. Otherwise, j keeps the lock and gossips an abortion message back to i. Algorithm 3 describes this procedure.

Algorithm 3: RTG-DS's Distributed Deadlock Detection Algorithm

```
Upon $j$ receiving $i$'s UP gossip message msg:
COPY(gossip, msg) ;
if DETECT(msg) = true then
    /* a distributed deadlock occurs */
    if $GUD_i > GUD_j$ then
        abort $j$;
        gossip.lsr ← msg.lsr − $abt_j$;
        give resource lock to $i$;
    else
        continue $j$'s execution;
        keep resource lock;
        gossip.lsr ← msg.lsr;
    gossip.round ← gossip.lsr/$\Psi$ ;
    gossip.c ← FANOUT(gossip.round);
    RTG_GOSSIP(gossip);
```

4. Algorithm Analysis

Let δ be the desired probability for delivering a message to its destination node within the gossip period Ψ. If the communication delay follows a Gamma distribution with a probability density function:

$$f(t) = \frac{(t - LCD)^{\alpha-1} e^{\frac{-(t-LCD)}{\beta}}}{\Gamma(\alpha)\beta^{\alpha}}, \quad t > LCD$$

where $\Gamma(\alpha) = \int_0^\infty x^{\alpha-1}e^{-x}dx$, $\alpha > 0$. Then, $\delta = \int_{LCD}^{t_b} f(t)\,dt$, $t > LCD$, where $t_b : D(t_b) = \delta$, and $D(t)$ is the distribution function. Note that LCD is the communication delay lower bound and $\Psi > t_b$.

We denote the message loss probability as $0 \le \sigma < 1$, and the probability for a node to fail during thread execution as $0 \le \omega < 1$. Let C denote the number of messages that a node sends during each gossip round (i.e., the fan out). We call a node *susceptible* if it has not received any gossip messages so far; otherwise it is called *infected*. The probability that a given susceptible node is infected by a given gossip message is:

$$p = \left(\frac{C}{n-1}\right)(1-\sigma)(1-\omega)\delta \tag{1}$$

Thus, the probability that a given node is not infected by a given gossip message is $q = 1 - p$. Let $I(t)$ denote the number of infected nodes after t gossip rounds, and $U(t)$ denote the number of remaining susceptible nodes after t rounds. Given i infected nodes at the end of the current round, we can compute the probability for j infected nodes at the end of the next round (i.e., $j - i$ susceptible nodes are infected during the next round). The resulting Markov Chain is characterized by the following probability $p_{i,j}$ of transitioning from state i to state j:

$$\begin{aligned} p_{i,j} &= P\left[I(t+1) = j | I(t) = i\right] \\ &= \begin{cases} \binom{n-i}{j-i}\left(1-q^i\right)^{j-i} q^{i(n-j)} & j \geqslant i \\ 0 & j < i \end{cases} \end{aligned} \tag{2}$$

The probability that the expected number of j nodes are infected after round $t+1$ is given by:

$$P\left[I(0) = j\right] = \begin{cases} 1 & j = 1 \\ 0 & j > 1 \end{cases} \tag{3}$$

$$P\left[I(t+1) = j\right] = \sum_{i \leqslant j} P\left[I(t) = i\right] p_{i,j} \tag{4}$$

Theorem 1. *RTG-DS probabilistically bounds thread time constraint satisfactions'.*

Proof. Let a thread will execute through m head nodes. The mistake probability p_{M_k} that a head node k cannot determine the thread's next destination head node after gossip completes at round t_{max} is given by:

$$\begin{aligned} p_{M_k} &= \left\{1 - P\left[I(t_{\max}) = \eta\right]\right\} \times \frac{1}{U(t_{\max})} \\ &= \left\{1 - \sum_{i \leqslant \eta} P\left[I(t_{\max-1}) = i\right] p_{i[\eta]}\right\} \times \frac{1}{U(t_{\max})} \end{aligned} \tag{5}$$

where η is the expected number of infected nodes after $t_{\max}$. This p_{M_k} is achieved when all nodes are not overloaded (consequently, RTG-DS's LSF-order being locally optimal, will feasibly complete all local sections).

Let w_k be the waiting time before section k's execution. w_k includes the section interference time, gossip time, and blocking time (we bound blocking time in Theorem 4). Now, X_k and X_m can be defined as:

$$X_k = \begin{cases} 1 & \text{If } w_k \le LS_{rk} - LCD \\ 0 & \text{Otherwise} \end{cases} \quad (6)$$

$$X_m = \begin{cases} 1 & \text{If } w_k \le LS_{rm} \\ 0 & \text{Otherwise} \end{cases} \quad (7)$$

If $X_k = 1$, the relative section can not only finish its execution, but it can also make a successful invocation. X_m is for the last destination node, so it does not consider the communication delay LCD. Thus, the probability for a distributable thread d to successfully complete its execution P_{S_d}, and that for a thread set D to complete its execution, P_{S_D}, is given by:

$$P_{S_d} = X_m \prod_{k \le m-1} (1 - p_{M_k}) X_k \quad P_{S_D} = \prod_{d \in D} P_{S_d} \quad (8)$$

□

Theorem 2. *The number of rounds needed to infect n nodes, t_n, is given by:*

$$t_n = \log_{C+1} n + \frac{\log n}{C} + o(1) \quad (9)$$

Proof. We skip the proof, due to page constraints. The proof is similar to [19], but we conclude the theorem under the assumption that the fan out C exceeds 1. □

Lemma 3. *A head node will expect its gossip message to be replied in at most $2t_n$ rounds, with a high (computable) probability.*

Proof. Suppose the next destination node N is the last node getting infected by the gossip message from a head node A. Thus, node A will take t_n rounds to gossip to node N. Suppose A is the last node to be infected by N's reply message, and it will take another t_n rounds. Thus, the worst case to determine the next destination node is $2t_n$ rounds. The probability can be computed using Equations 3 and 5. Since the fan out number C can be adjusted, we can get a required probability by modifying C into a proper value. □

Theorem 4. *If a thread section is blocked by another thread section on a different node, then its blocking time under RTG-DS is probabilistically bounded.*

Proof. Suppose section i is blocked by section j whose head is now on a different node. According to Theorem 2, it will take section i at most t_{n_i} time rounds to gossip an utility propagation (UP) message to section j's head node.

After j's head node receives i's UP message, RTG-DS will compare i's GUD with j's. If $GUD_i > GUD_j$, then j must grant the lock to i as soon as possible. According to Algorithm 2, the handler will deal with j's head within $\texttt{min}(abt_j, er_j)$. According to Lemma 3, i's head will expect a reply from j after at most t_{n_i} time rounds. If $t_{n_i} - \texttt{min}(abt_j, er_j) \ge LCD$, then j can reply and grant lock to i at the same time. Thus, i's blocking time bound $b_{i,j} = 2t_{n_i}$. Otherwise, j must first reply to i. Since i's head needs at least LCD gossip time to continue execution, the blocking time is at most $LS_{ri} - LCD$. Thus, if $(LS_{ri} - LCD) - t_{n_i} - \texttt{min}(abt_j, er_j) \ge LCD$, $b_{i,j} = LS_{ri} - LCD$. If not, i has to be aborted because there is not enough time to grant the lock. Under this condition, RTG-DS aborts i, and $b_{i,j} = 2t_{ni}$, since j need not respond any more after the first reply to i. If $GUD_i \le GUD_j$, then j will not grant i the lock until it finishes necessary execution. Thus, $b_{i,j} = LS_{ri} - LCD$.

The probability of the blocking time bound is induced by RTG-DS's gossip process. It can be computed using (3) and (5), and a desired probability can be obtained by adjusting the fan out C. □

Theorem 5. *RTG-DS probabilistically bounds deadlock detection and notification times.*

Proof. As shown in Fig. 5, there are two possible situations: 1) deadlock happens when section i requires resource R2, or 2) when section j's REQ R1 message arrives at Node 2.

Let $GUD_i > GUD_j$. Under the first condition, i will check the necessary time for deadlock solution, which is denoted as ds_{i2}. Let $LS_{r_i 2}$ be the remaining local slack time of section i on Node 2, $t_{n_i 2}$ be the time rounds needed by i to gossip to Node 1 in order to finish i on time, and abt_{j1}, abt_{j2} be the needed abortion time of section j on Node 1 and 2, respectively.

Then, $ds_{j2} = abt_{j2}$, if no LIFO-ordered abortion is necessary from node 1 to node 2; otherwise $ds_{j2} = abt_{j1} + abt_{j2} + 2t_{ni2}$. By LIFO-ordered abortion, the last executed section is the first one that is aborted.

Under the second condition, deadlock happens when j's REQ message arrives at Node 2. Now, $ds_{j2} = t_{nj1}$, if $t_{nj1} - abt_{j2} \ge LCD$, or if $t_{nj1} - (abt_{j1} + abt_{j2} + 2t_{ni2}) \ge LCD$. Otherwise, $ds_{j2} = \max(t_{nj1} + abt_{j1} + t_{ni1}, abt_{j2})$.

Thus, if $ds_{j2} \le LS_{ri2} - LCD$, the scheduler will resume i. Otherwise, it will abort i since i won't have necessary remaining local slack time for gossiping.

The analysis is similar if $GUD_i > GUD_j$. The probabilistic blocking time bound is induced by RTG-DS's gossiping. It can be computed using (3), and a desired probability can be obtained by adjusting fan out C. □

Theorem 6. *RTG-DS probabilistically bounds failure-exception notification times for aborting partially executed sections of failed threads.*

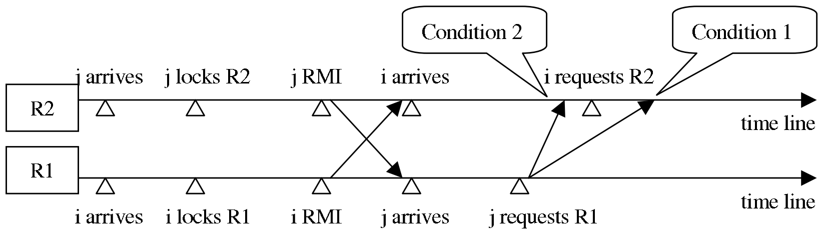

Figure 5. Example Distributed Deadlock

Proof. From Lemma 6 in [10], we obtain the failure-exception notification time f_n as follows: $f_n = 3t_n$, if no LIFO-ordered abortion is necessary from node m to node n. Otherwise, $f_n = 3t_n + \sum_{i=m,\ldots,n} t_{ni}$. $\square$

5. Simulation Studies

We evaluate RTG-DS's effectiveness by comparing it with "RTG-DS/DASA" — i.e., RTG-DS with DASA as the section scheduler — as a baseline. We do so because DASA exhibits very good performance among most UA scheduling algorithms [20]. We use uniform distribution to describe section inter-arrival times, section execution times, and termination times of a set of distributable threads. All threads are generated to make invocations through the same set of nodes in the system. However, the relative arrival order of thread invocations at each node may change due to different section schedules on nodes. Thus, it is quite possible that a thread may miss its termination time because it arrives at a destination node late.

A fixed number of shared resources is used in the simulation study. The simulations featured four (one on each node) and eight (two on each node) shared resources, respectively. Each section probabilistically determines how many resources it needs. Each time a resource is acquired, a fraction (following uniform distribution) of the section's remaining execution time is assigned to it.

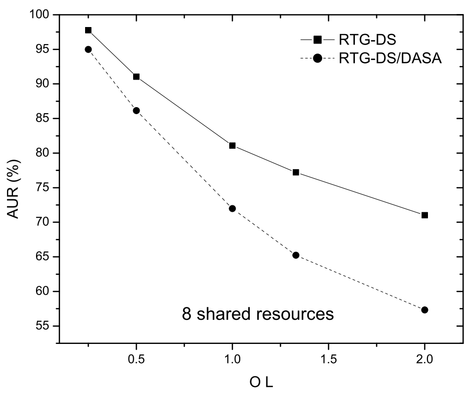

Figure 6. AUR in a 8-Resource-System Under RTG-DS and RTG-DS/DASA

We measure RTG-DS's performance using the metrics of Accrued Utility Ratio (AUR), Termination time Meet Ratio (TMR) and Offered Load (OL) in a 100-node system. AUR is the ratio of the total accrued utility to the maximum possible total utility, TMR is the ratio of the number of threads meeting their termination times to the total thread releases in the system, and OL is the ratio of a section's expected execution time to the expected section inter-arrival time. Thus, when OL < 1.0, a section will most possibly complete its execution before the next section arrives; when OL > 1.0, system will have long-term overloads.

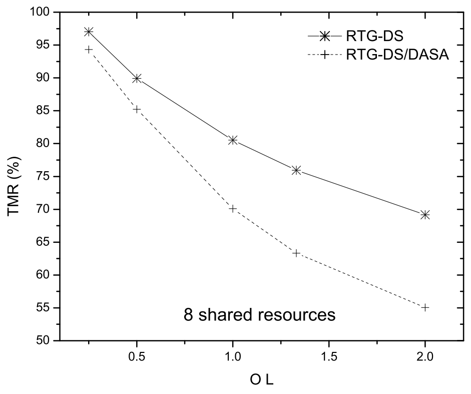

Figure 7. TMR in a 8-Resource-System Under RTG-DS and RTG-DS/DASA

Note that RTG-DS uses the novel techniques that we have presented here including $\rho_R(t)$, GUD and PUD, selecting LSF section, utility propagation, and distributed deadlock resolution. RTG-DS/DASA does not use any of these, but only follows RTG-DS in the gossip-based searching of next destination nodes.

Figures 6 and 7 show the results for the eight-resource system. From the figure, we observe that RTG-DS gives much better performance than RTG-D/DASA. Further, when OL is increased, both algorithms' AUR and TMR decrease. We observe consistent results for the four-resource case, but omit them here for brevity.

In Fig. 8, as discussed in Section 3.3.3, we observe that under any OL, RTG-DS has a smaller variance of remaining local slack time than RTG-DS/DASA, because it first executes the least-slack section instead of the earliest local termination time section. By this way, though sections' mean value of remaining local slack time after execution is almost the same, RTG-DS gives sections with less local slack time more chances to finish their gossip process, and thus more chances

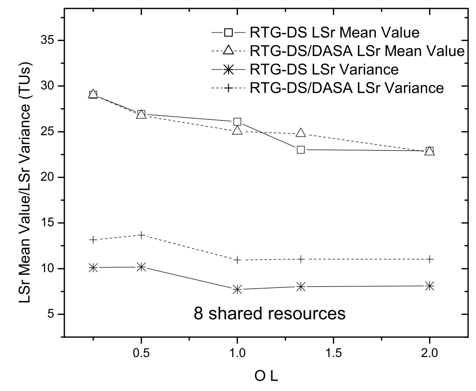

Figure 8. Remaining Local Slack Time Under RTG-DS and RTG-DS/DASA (Mean, Variance)

to find their next destination nodes.

6. Conclusions and Future Work

In this paper, we present a gossip-based algorithm called RTG-DS, for scheduling distributable threads under dependencies in ad hoc networks. The algorithm uses gossip-based communication for (a) propagating thread scheduling parameters, (b) determining successive nodes for feasible thread execution, (c) speeding-up the execution of blocking threads, and (d) detecting and resolving deadlocks. RTG-DS constructs local thread section schedules by exploiting thread slack in a way that enhances time available for gossiping.

We prove that RTG-DS probabilistically bounds thread blocking times and deadlock detection and notification times, thereby probabilistically bounding thread time constraint satisfactions'. Further, we show that the algorithm probabilistically bounds failure-exception notification times for aborting partially executed sections of failed threads. Our simulation studies validate the algorithm's effectiveness.

Immediate directions for extending our work include allowing node anonymity, unknown number of thread sections, and non-step TUFs. Other long term goals include extending RTG-DS to arbitrary graph-shaped, multi-node, causal control and/or data flows.

References

[1] T. Abdelzaher et al. A feasible region for meeting aperiodic end-to-end deadlines in resource pipelines. In *ICDCS*, pages 436–445, 2004.

[2] J. Anderson and E. D. Jensen. The distributed real-time specification for java: Status report. In *JTRES*, 2006. Available: `http://www.real-time.org/docs/jtres06/jtres06.pdf`.

[3] F. Baker. An outsider's view of manet. Internet-Draft, Work In Progress draft-baker-manet-review-01.txt, IETF Network Working Group, March 2002.

[4] A. Bestavros and D. Spartiotis. Probabilistic job scheduling for distributed real-time applications. In *IEEE Works. on Real-Time Applications*, May 1993.

[5] R. Bettati. *End-to-End Scheduling to Meet Deadlines in Distributed Systems.* PhD thesis, UIUC, 1994.

[6] K. P. Birman, M. Hayden, et al. Bimodal multicast. *ACM TOCS*, 17(2):41–88, 1999.

[7] CCRP. Network centric warfare. `http://www.dodccrp.org/html2/research_ncw.html`. Last accessed, May 2006.

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

[9] K. Han et al. Probabilistic, real-time scheduling of distributable threads under dependencies in mobile, ad hoc networks. Available: `http://www.real-time.ece.vt.edu/rtgd.pdf`, 2006.

[10] K. Han, B. Ravindran, and E. D. Jensen. Real-time gossip: Probabilistic, distributed real-time scheduling in ad hoc networks. Available: `http://www.real-time.ece.vt.edu/rtg.pdf`, 2006.

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

[12] D. Johnson et al. Dsr: The dynamic source routing protocol for multihop wireless ad hoc networks. In C. E. Perkins, editor, *Ad Hoc Networking*, chapter 5, pages 139–172. Addison-Wesley, 2001.

[13] H. Li et al. Bar gossip. In *OSDI*, November 2006.

[14] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms.* PhD thesis, Virginia Tech, 2004.

[15] B. S. Manoj et al. Real-time traffic support for ad hoc wireless networks. In *IEEE ICON*, pages 335 – 340, 2002.

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

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

[18] OMG. Real-time corba 2.0: Dynamic scheduling specification. Technical report, OMG, September 2001. Final Adopted Specification, `http://www.omg.org/docs/ptc/01-08-34.pdf`.

[19] B. Pittel. On spreading a rumor. *SIAM J. Appl. Math.*, 47(1), 1987.

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

[21] K. Romer. Time synchronization in ad hoc networks. In *MobiHoc*, pages 173–182, 2001.

[22] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[23] N. R. Soparkar, H. F. Korth, and A. Silberschatz. *Time-Constrained Transaction Management.* Kluwer Academic Publishers, 1996.

[24] J. Sun. *Fixed-Priority End-To-End Scheduling in Distributed Real-Time Systems.* PhD thesis, UIUC, 1997.

[25] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 50(2-3), 1994.

[26] N. Wang and C. Gill. Improving real-time system configuration via a qos-aware corba component model. In *HICSS*, page 10, 2004.