Probabilistic, Real-Time Scheduling of Distributable Threads Under Dependencies 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 that are subject to dependencies (e.g., due to mutual exclusion constraints) in ad hoc networks, in the presence of node and link failures, message losses, and dynamic node joins and departures. We present a gossip-based distributed scheduling algorithm, called RTG-D. We prove that thread blocking times under RTG-D are probabilistically bounded, thereby probabilistically bounding thread time constraint satisfactions'. Our simulation results validate RTG-D's effectiveness.**

I. Introduction

Causally-chained, multi-node sequential behaviors (e.g., a sequence of multi-node, remote method executions) are common in many distributed applications. Since partial failures are the common case rather than the exception in many of them, applications typically desire those behaviors to exhibit application-specific, end-to-end integrity properties. Real-time distributed applications also desire (application-specific) end-to-end timeliness properties, besides integrity.

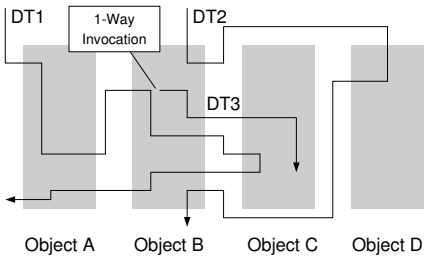


Fig. 1. Distributable Threads

An abstraction for programming such causal behaviors and for enforcing end-to-end properties on them is *distributable threads* [1], [2]. A distributable thread is a single thread of execution with a globally unique ID that 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 movement within/among objects and nodes. Fig. 1 shows the execution of three distributable threads [3]. In the rest of the paper, we will refer to distributable threads as *threads*, unless qualified.

A thread carries its execution context as it transits node boundaries, including its scheduling parameters (e.g., time constraints, execution time), ID, security credentials, and any application data. The propagated 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), and for scheduling threads to optimize system-wide timeliness. Thus, threads constitute the abstraction for concurrency and scheduling.

In terms of providing direct support for causal multi-node behaviors, threads can be viewed as at a higher-level of abstraction than models such as publish/subscribe (P/S) [4]. With P/S, a causal sequence can also arise—e.g., publication of topic A depends on subscription of topic B; publication of B, in turn, depends on subscription of topic C, and so on. Enforcing end-to-end properties (timeliness, integrity) on such a causal P/S chain will require similar context-based mechanisms as that of threads. Thus, the problem of enforcing end-to-end properties on such causal chains — whether programmed using threads or P/S — is conceptually similar.

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), 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., [5]) is their relatively long thread execution time magnitudes—e.g., milliseconds to minutes. Despite the uncertainties, such applications desire the strongest possible assurances on end-to-end thread timeliness behavior. Stochastic 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 between them. For example, a thread A may lock a resource on a node and may make a remote invocation, carrying the lock with it. Another thread B may later arrive at the node and request the lock. Thread B is now blocked by thread A, until A returns back to the node from its remote invocation and releases the lock. The time interval from B's lock request till A's release of the lock is the blocking time that B suffers from A. Unbounded blocking time can be antagonistic to system-wide timeliness optimization—e.g., thread B may have a greater urgency than thread A.

In this paper, we present an algorithm called *Real-Time Gossip algorithm for Dependent threads* (or RTG-D) that provides assurances on such blocking times in ad hoc networks. RTG-D exploits the randomness of gossip protocols, which results in their reliability and robustness in ad hoc networks (see [6] and the references therein) to counter application/network-induced uncertainties in our context. In doing so, the algorithm answers the fundamental question of how to design a gossip protocol that yields assurances on thread blocking times. We prove that thread blocking times and thread time constraint satisfactions' are probabilistically bounded under RTG-D. Our simulation studies verify the algorithm's effectiveness.

End-to-end real-time scheduling has been studied in the past (e.g., [2], [7]–[12]), but these are limited to fixed infrastructure networks. End-to-end timing assurances in ad hoc networks are considered in [13]–[16]. However, none of these works consider dependent threads, which is precisely what our work does. Thus, the paper's contribution is the RTG-D that provides probabilistic end-to-end timing assurances for dependent distributable threads in ad hoc networks. We are not aware of any other efforts that solve the problem solved by RTG-D.

The rest of the paper is organized as follows: In Section II, we discuss our models and state the algorithm objectives. Section III presents RTG-D. We analyze RTG-D in Section IV. In Section V, we report our simulation studies. We conclude the paper and identify future work in Section VI.

II. Models and Algorithm Objectives

Distributable Threads. Distributable threads [1]–[3] 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*.

A thread's initial segment is called its *root* and its most recent segment is called its *head*. A thread's head 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.

Execution time estimates of the sections of a thread are known when the thread arrives at the respective nodes. The time estimate includes that of the section's normal code as well as its exception handler code, and can be violated at run-time (e.g., due to context dependence), causing CPU overloads at the node.

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 assumed to be known. The ID of the nodes hosting the objects, and the sequence of the thread invocations are assumed to be unknown at thread creation time, as nodes may dynamically fail, or join, or leave the network.

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

Timeliness Model. Each thread's time constraint is specified using the time/utility function (or TUF) model [17]. A TUF specifies the utility of completing a thread as a function of its completion time. Fig. 2 shows example downward "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 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.

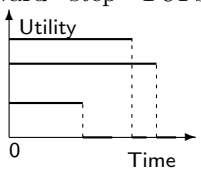


Fig. 2. Step TUFs

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

If a thread has not completed by its termination time, then a failure-exception is raised, and exception handlers are released for aborting all partially executed sections (for releasing system resources). The handlers' time constraints are specified using TUFs.

Resource Model. Thread sections can access non-CPU resources (e.g., disks, NICs) located at their nodes during their execution, which in general, are serially reusable. Similar to fixed-priority resource access protocols [18] and that for TUF algorithms [19], [20], 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. We assume that a thread explicitly releases all granted resources before the end of its execution.

All resource request/release pairs are assumed to be confined within nodes. Thus, a thread cannot request (and 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 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., the resources that will be needed,

and the order of accessing them are all assumed to be a-priori unknown. Further, threads can have precedence constraints. For example, a thread T_k can become eligible for execution only after a thread T_l has completed, because T_k may require T_l's results. As in [19], [20], we allow such precedences to be programmed as resource dependencies.

System Model. The network consists of a set of nodes, denoted $N = \{n_1, n_2, n_3, \ldots\}$, communicating through bidirectional wireless links. A basic unicast routing protocol such as DSR [21] is assumed to be available for packet transmission between nodes. 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 [22]. Nodes may dynamically join or leave the network. We assume that the network communication delay follows some non-negative probability distribution such as the Gamma distribution. Nodes may fail by crashing, links may fail transiently or permanently, and messages may be lost, all arbitrarily.

Objectives. Our goal is to design a thread scheduling algorithm with probabilistic blocking time bounds (i.e., a thread's probabilistically-satisfied blocking time bound must be computable) and termination-time satisfactions (i.e., the probability for a thread to satisfy its termination time must be computable). Further, we desire to maximize the total thread accrued utility.

III. The RTG-D Algorithm

We first overview RTG-D's operation at a high-level. When a thread arrives at a node, RTG-D decomposes the thread's end-to-end TUF into a set of local TUFs, one for each of the remaining remote invocations. Local TUFs are used for scheduling thread sections.

When a thread completes its execution on a node, RTG-D starts a finite series of synchronous gossip rounds. During each round, the node randomly selects a set of neighbors and queries whether it can execute the thread's next invocation, satisfying its local TUF. The number of gossip rounds, and the number of neighbor nodes (i.e., the "fan-out") are derived from the local slack, as they directly affect the time incurred for gossip, and thereby affect the next invocation's available slack for execution.

When a node receives a gossip message, it checks whether it hosts the requested invocation, and can feasibly execute it. If so, it replies back to the node where the gossip originated. If not, the node starts a set of gossip rounds (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 that node. 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 thread exception handlers (for aborting partially executed thread portions).

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

A. Building Local Scheduling Parameters

RTG-D decomposes the thread's end-to-end TUF based on the execution time estimates of the thread sections and the thread's 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 remaining 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 El_i. RTG-D 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-D determines the local slack for a thread in a way that allows the 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-D determines the next node for executing the thread's next section, through a set of gossip rounds. The network communication delay incurred by RTG-D 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 $LX_{i,j} = t + El_i + LS_{i,j}$. The local termination time is used by RTG-D to test for schedule feasibility, while constructing local thread section schedules (we discuss this in Section III-C).

B. Determining Thread's Next Node

Once the execution of a section completes on a node, RTG-D 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-D 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 a thread T_i's local section complete on node n_j 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-D determines the number of gossip rounds before $LX_{i,j}$ as $round = LSr_{i,j}/\Psi$. RTG-D also determines the number of messages that must be sent during each gossip round, called *fan out*, for determining the next node.

RTG-D 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-D algorithms on head node and intermediate node can be found in [14].

C. Constructing Section Schedules

RTG-D constructs local section schedules with the goal of maximizing the total utility accrued by all threads, while respecting thread dependencies. Thus, local thread sections must be scheduled to meet their local termination times. Further, it must maximize their remaining local slack times, which will increase the gossiping time for the subsequent sections, and thus the likelihood for the (respective) threads to complete before their global termination times. The section scheduling algorithm is derived from the Dependent Activity Scheduling Algorithm (DASA) [19].

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. 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. 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. Before aborting a section, the resources held by the section are released and returned to consistent states.

After handling deadlocks, the algorithm examines sections in the order of non-increasing *potential utility densities* (or PUDs). The PUD of a section is the ratio of the expected section utility to the remaining execution time of the section and its dependents, and thus measures the section's return on "investment." Thereafter, the algorithm inserts each section and its dependents into a tentative schedule that is ordered by section slack (earliest slack first). The insertion is done by respecting the sections' dependencies.

After insertion, RTG-D checks the schedule's feasibility with respect to satisfying all section termination times. If infeasible, the inserted section and its dependents are rejected. The algorithm repeats the process until all sections are examined, and selects the section with the least slack for execution, to allow for greater gossiping time for determining the thread's next node.

If the selected section for execution is a remote section (because it holds a locally requested lock), then the algorithm speeds-up the remote section's execution by propagating the utility of the local dependents.

IV. 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, $0 \leq \sigma < 1$, and the probability for a node to fail during thread execution as $0 \leq \omega < 1$. 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, the node 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:

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

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

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

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

We established Theorem 1 and Lemma 2 in [13]:

Theorem 1: The number of rounds needed to infect n nodes, t_n, is given by:

$$t_n = \log_{C+1} n + \frac{\log n}{C} + o(1) \tag{5}$$

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

Theorem 3: If a thread section is blocked by another thread section on a different node, then its blocking time under RTG-D is probabilistically bounded.

Proof: Suppose section i is blocked by section j whose head is now on a different node. According to Theorem 1, it will take section i at most t_{n_i} time rounds to gossip a message to section j's head node.

After j's head node receives i's message, RTG-D will compare i's GUD with j's. If $GUD_i > GUD_j$, then j should give the lock to i as soon as possible, and the handler should deal with j's head within `min(`abt_j`,`er_j`)`, where abt_j is the time used to abort section j, and er_j is section j's remaining execution

time. According to Lemma 2, i's head will expect a reply from j after at most t_{n_i} time rounds. If $t_{n_i} - \texttt{min}(abt_j, er_j) \geq LCD$, then j can reply and give resource lock to i at the same time. Thus, i's blocking time bound $b_{i,j} = 2t_{n_i}$. Otherwise, j should 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 $(LS_{ri} - LCD) - t_{n_i} - \texttt{min}(abt_j, er_j) \geq LCD$, $b_{i,j} = LS_{ri} - LCD$. If not, i has to be aborted because there is not enough time to give back the resource lock. Under this condition, RTG-D aborts i, and $b_{i,j} = 2t_{ni}$, since j need not respond any more after the first reply to i. If $GUD_i \leq GUD_j$, then j will not give i the resource lock until it finishes necessary execution. Thus, $b_{i,j} = LS_{ri} - LCD$.

The probability of the blocking time bound on a given resource is induced by RTG-D's gossip process. It can be computed by recursively using (3) and (4), and a desired probability can be obtained by adjusting the fan out C. ■

Theorem 4: RTG-D 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:

$$\begin{aligned} p_{M_k} &= \{1 - P[I(t_{\max}) = \eta]\} \times \frac{1}{U(t_{\max})} \\ &= \left\{1 - \sum_{i \leqslant \eta} P[I(t_{\max -1}) = i]\, p_{i[\eta]}\right\} \times \frac{1}{U(t_{\max})} \end{aligned} \tag{6}$$

where η is the expected number of infected nodes after $t_{\max}$.

Let w_k be the waiting time before section k's execution, w_k^i be the waiting time on execution interference by other thread sections, $w_k^b(x,j)$ be the waiting time on resource x's blocking caused by another thread section j, M be the number of resources, and N be the number of threads. Then:

$$w_k = \sum_{x \leqslant M} \sum_{j \leqslant N} w_k^b(x,j) + w_k^i \tag{7}$$

where $\{w_k, w_k^i, w_k^b(x,j)\} \in [0, LS_{rk} - LCD]$ if node k is not the last destination node; or $\{w_k, w_k^i, w_k^b(x,j)\} \in [0, LS_{rm}]$ if it is the last one.

Let's assume that w_k^i will never exceed its upper bound. Then, we have:

$$\sum_{x \leqslant M} \sum_{j \leqslant N} w_k^b(x,j) = w_k - w_k^i \tag{8}$$

Thus, if the thread section can successfully complete on destination node k, then $w_k^b(x,j)$ should follow (8).

Let $P_b(j,k,x)$ be the probability that section j will block section k on resource x, and let $p_b(j,k,x)$ be the probability that section j will block section k on resource x within the required time bound given in (8). Now, X_k can be defined as:

$$X_k = \min_{x \leqslant M} \left(\prod_{j \leqslant N} P_b(j,k,x) \times p_b(j,k,x) \right) \tag{9}$$

X_k represents the probability that the relative section can not only finish its execution, but also can make a successful invocation. Consider the worst case: every other thread section will block section k. Thus, the lower bound of X_k is:

$$X_k = \min_{x \leqslant M} \left(\prod_{j \leqslant N} p_b(j,k,x) \right) \tag{10}$$

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:

$$P_{S_d} = \prod_{k \leq m} (1 - p_{M_k}) X_k \qquad P_{S_D} = \prod_{d \in D} P_{S_d} \tag{11}$$

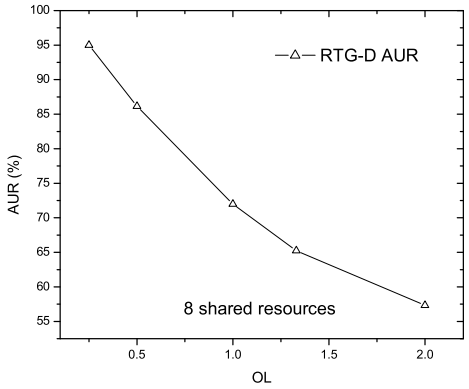
■

V. Simulation Studies

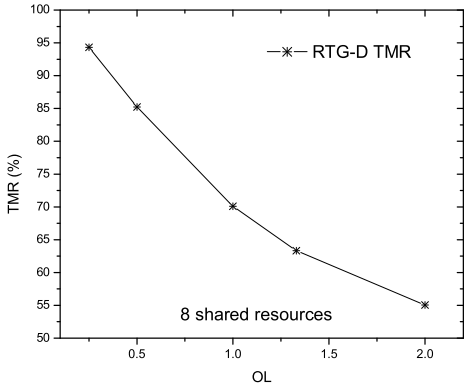
We conducted simulation studies to evaluate RTG-D's performance. We used uniform distribution to describe the inter-arrival times, section execution times, and termination times of a set of threads. All threads are generated to make invocations through the same set of nodes in the system. However, the 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 was 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 of these resources it must acquire to successfully complete execution. Once the number of resources has been decided, the exact identities of shared resources that will be needed are chosen randomly. Each time a resource is acquired, a fraction of the computation time remaining in the section elapses before the next resource is requested. This fraction is drawn from a uniform probability distribution.

We measure RTG-D'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 number of thread releases, and OL is the ratio of the total expected execution time of all thread sections to the expected thread inter-arrival time. Thus, when OL < 1.0, threads will complete their execution before they arrive again; when OL > 1.0, system will have long-term overloads.



(a) RTG-D AUR in 8-Resource-System



(b) RTG-D TMR in 8-Resource-System

Fig. 3. AUR and TMR in a 8-Resource-System Under RTG-D

Fig. 3 shows the results for the 8-resource system. From Fig. 3(a), we observe that RTG-D has excellent performance when OL is small (AUR = 95% when OL = 0.33 and AUR = 86% when OL = 0.5). AUR is decreased when OL is larger. However, because RTG-D can filter ineligible threads before execution, it still gets good performance when the system has long-term overloads. For instance, AUR is above 60% when OL = 1.33, which means that there is no possibility for 1/3 of total threads to be finished on time. We also observe RTG-D's similar performance in Fig. 3(b).

VI. Conclusions and Future Work

In this paper, we present a gossip-based algorithm called RTG-D, for scheduling distributable threads that are subject to dependencies in ad hoc networks, in the presence of arbitrary node/link failures, message losses, and varying node membership. The algorithm uses gossip-based communication for (a) propagating thread scheduling parameters, (b) determining successive nodes for feasible thread execution, and (c) speeding-up the execution of blocking threads. RTG-D constructs local thread section schedules using propagated and locally constructed scheduling parameters. We prove that RTG-D probabilistically bounds thread time constraint satisfactions in the presence of dependencies. Our simulation studies validate RTG-D's effectiveness.

Some example directions for extending our work include allowing node anonymity, studying standard deviations for experiment results, multiple nodes to host same thread sections, unknown number of thread invocations, and non-step TUFs.

References

[1] 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`.

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

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

[4] OMG, "Data distribution service for real-time systems, v1.1," 2005, formal/2005-12-04.

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

[6] H. Li, A. Clement, et al., "Bar gossip," in *USENIX OSDI*, November 2006.

[7] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing & Microprogramming*, vol. 50, no. 2-3, 1994.

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

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

[10] R. Bettati, *End-to-End Scheduling to Meet Deadlines in Distributed Systems*, Ph.D. thesis, UIUC, 1994.

[11] T. F. Abdelzaher and K. G. Shin, "Combined task and message scheduling in distributed real-time systems," *TPDS*, vol. 10, no. 11, pp. 1179–1191, November 1999.

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

[13] 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.

[14] K. Han et al., "Scheduling distributable real-time threads in dynamic ad hoc networks with probabilistic failure recovery times," Available: `http://www.real-time.ece.vt.edu/rtg-fd.pdf`, 2006.

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

[16] N. Wang and C. Gill, "Improving real-time system configuration via a qos-aware corba component model," in *HICSS*, 2004, p. 10.

[17] E. D. Jensen et al., "A time-driven scheduling model for real-time systems," in *RTSS*, Dec. 1985, pp. 112–122.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

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

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

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

[22] K. Romer, "Time synchronization in ad hoc networks," in *MobiHoc*, 2001, pp. 173–182.