Probabilistic, Real-Time Scheduling of Distributable Threads Under Dependencies in 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 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 Real-Time Gossip algorithm for Dependencies (or RTG-D). RTG-D builds upon our prior algorithm called RTG, that schedules distributable threads without dependencies in ad hoc networks. We prove that thread blocking times under RTG-D are probabilistically bounded, thereby probabilistically bounding thread time constraint satisfactions’ under the algorithm. Further, we prove that RTG-D probabilistically bounds deadlock detection and notification times, and failure-exception notification times for aborting partially executed sections of failed threads. Our simulation results validate RTG-D’s effectiveness.

I. Introduction

Many distributed systems can be reasoned about in terms of asynchronous concurrent sequential flows of execution within and among objects. The distributable thread programming model of OMG’s recent Real-Time CORBA 1.2 standard (abbreviated here as RTC2) [1] and Sun’s upcoming Distributed Real-Time Specification for Java (DRTSJ) standard [2] directly provides that as a first-class abstraction. Distributable threads first appeared in the Alpha OS [3] and later in the MK7.3 OS [4].

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 movement within/among objects and nodes. In the rest of the paper, we will refer to distributable threads as threads except as necessary for clarity.

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 resources (e.g., CPU, I/O), and for scheduling threads to optimize system-wide timeliness. Thus, threads constitute the abstraction for concurrency and scheduling. Fig. 1 shows the execution of threads [1].

In this paper, we consider scheduling threads that are subject to dependencies (e.g., due to mutual exclusion constraints) in ad hoc network systems, in the presence of application- and network-induced uncertainties. By ad hoc networks, we mean those without a fixed network infrastructure, including mobile and wireless networks. Application-induced uncertainties typically include transient and sustained resource overloads (due to context-dependent thread execution times), arbitrary thread arrivals, and arbitrary node and link failures. Network-induced uncertainties often include transient and permanent link failures, and varying packet drop rate behaviors and message losses (some of these can also be application-induced). Another important distinguishing feature of most of these systems that are of interest to us is their relatively long thread execution time magnitudes, compared to conventional real-time subsystems—e.g., on the order of milliseconds to minutes.

Despite the application/network-induced uncertainties, such systems desire the strongest possible assurances on thread timeliness behavior. Stochastic assurances (e.g., meeting deadlines with 80% probability)
are often appropriate. Example systems that motivate our work (from the defense domain) include sensor applications such as Active Electronically Steerable Array (AESA) radars [5], combat platform applications such as surveillance aircraft [6], and large-scale enterprise applications such as network-centric warfare [7].

In [8], we presented a gossip-based distributed scheduling algorithm called Real-Time Gossip algorithm (or RTG) that achieves these objectives. RTG uses randomness to “fight” uncertainties — an idea that is central to the class of gossip protocols, which has been used to solve a variety of problems—e.g., data management [9], reliable multicast [10,11], failure detection [12], dissemination [13], aggregation [14].

In this paper, we build upon RTG, and present an algorithm called RTG-D (or RTG for Dependencies) that allows dependencies among threads including those that are caused by mutual exclusion and precedence constraints. We consider a resource model, where threads may concurrently share node-local, non-CPU resources (e.g., I/O, disk, NIC) under mutual exclusion constraints using lock-based synchronizers.

We prove that thread blocking times — the length of time during which threads wait for their requested lock — are probabilistically bounded under RTG-D. Consequently, we prove that RTG-D’s thread time constraint satisfactions’ are probabilistically bounded. Further, we prove that RTG-D probabilistically bounds dead-lock detection and notification times, and failure-exception notification times for aborting partially executed portions of failed threads. Our simulation studies verify the algorithm’s effectiveness.

End-to-end real-time scheduling has been studied in the past (e.g., [3,15–20]), but they are limited to fixed infrastructure networks. This is overcome for the first time by the RTG for ad hoc networks [8]. However, [8] is limited to independent threads. 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

A. 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. 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. 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. A section’s first segment results from an invocation from another node, and its last segment performs a remote invocation.

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

B. Timeliness Model

Each thread’s time constraint is specified using a time/utility function (or TUF) [21]. A TUF specifies the utility of completing a thread as a function of its completion time. Fig. 2 shows 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; the most urgent can be the most important, and vice versa.

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’s termination time is reached and its execution has not been completed, a failure-exception is raised, and exception handlers are released for aborting all partially executed thread sections (for releasing system resources). The handlers’ time constraints are also specified using TUFs.

C. 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 [22] and that for TUF algorithms [23,24], 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 [23,24], we allow such precedences to be programmed as resource dependencies.

D. 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 [25] 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). We assume that node clocks are synchronized using an algorithm such as [26]. 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.

Each object transited by threads is assumed to be uniquely hosted by a node. Threads may be created at arbitrary times at a node. Upon creation, threads are assumed to present: (1) the number of objects on which they will make invocations (the identity of those objects and their hosting nodes, and the sequence of those invocations are assumed to be unknown at creation time); (2) execution time estimates of each of the sections (the time estimates include 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 CPU overloads); and (3) end-to-end TUFs.

E. Objectives

Our goal is to design an algorithm that can schedule threads with probabilistic 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. Furthermore, the time needed to detect a distributed deadlock and notify sections of a thread that must be aborted to resolve the deadlock must be bounded (so that exception handlers for aborting the sections can be released).\(^1\) Moreover, the time needed to notify partially executed sections of a failed thread (again, for releasing handlers for aborting the sections) must also be bounded.

\(^1\)We consider a deadlock detection and resolution strategy (as opposed to deadlock avoidance or prevention) precisely because of our assumption — which resources will be needed by which threads, and in what order is not a-priori known.
III. The RTG-D Algorithm

As mentioned before, RTG-D builds upon RTG. Many aspects of RTG-D (e.g., constructing local scheduling parameters, failure detection) are the same as that of RTG. In describing RTG-D, we describe the entire algorithm for completeness. In doing so, we only summarize the overlapping aspects and detail the unique aspects. 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 a thread’s end-to-end TUF based on the thread’s execution time estimates and 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 $E_{r_i}$, the total remaining slack time be $S_{r_i}$, the number of remaining thread sections (including the local section on $n_j$) be $N_{r_i}$, and the execution time of the local section be $E_{l_i}$. RTG-D computes a local slack time $LS_{i,j}$ for $T_i$ as $LS_{i,j} = S_{r_i} / (N_{r_i} - 1)$, if $N_{r_i} > 1$; $LS_{i,j} = S_{r_i}$, if $0 \leq N_{r_i} \leq 1$.

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 $LTT_{i,j} = t + E_{l_i} + LS_{i,j}$. 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.

Let the execution of a thread $T_i$’s local section complete on node $n_j$ at time $t_e$. $T_i$’s remaining local slack time is given by $LS_{r_{i,j}} = LTT_{i,j} - t_e$. Note that $LS_{r_{i,j}}$ is not always equal to $LS_{i,j}$, due to the interference that the thread section suffers from other sections on the node. Thus, $LS_{r_{i,j}} \leq LS_{i,j}$. With a gossip period $\Psi$, RTG-D determines the number of gossip rounds before $LTT_{i,j}$ as $\text{round} = LS_{r_{i,j}} / \Psi$. RTG-D also determines...
the number of messages that must be sent during each gossip round, called fan out \( C \), 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 [8].

C. Constructing Section Schedules

RTG-D constructs local section schedules with the goals of (a) maximizing the total utility, (b) maximizing the number of local termination times that are met, and (c) increasing the likelihood for global termination times to be met, while respecting dependencies. The algorithm is derived from the DASA algorithm [23].

We first overview the scheduling algorithm. 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, 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. 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 (least 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 the schedule is 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.

If a remote section holds a locally requested lock, then the algorithm speeds-up the remote section’s execution by propagating the utility of its local dependents.

We now explain key steps of the algorithm in greater detail.

C.1 Computing Potential Utility Density

RTG-D examines sections in non-increasing PUD-order to accumulate as many high PUD-sections into the schedule as possible, and thereby maximize the total utility as much as possible. Section \( i \)’s PUD is:

\[
PUD_i = \begin{cases} 
0 & \text{If } i \text{ is aborting} \\
\frac{U_i + U(Dep(i))}{c_i + c(Dep(i))} & \text{Otherwise}
\end{cases}
\]  

(1)

Here, \( U_i \) is the utility of section \( i \), and \( Dep(i) \) is the set of thread sections on which section \( i \) depends on. Section \( i \) is said to be dependent upon a section \( j \), if \( i \) needs a resource that is currently held by \( j \). Note that section \( i \)’s PUD can dynamically change, since \( i \)’s remaining execution time \( c_i \) and \( Dep(i) \) may change over time.

C.2 Determining Schedule Feasibility

RTG-D determines a node’s processor load \( \rho_R \) 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 \(i\). \(\rho_R\) in time interval \([t, d_i]\) is given by:

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

where \(c_k(t)\) is the remaining execution time of section \(k\) with \(d_k \leq d_i\), and \(\text{LCD}\) is the communication delay lower bound. Different from computing the processor load for a one-node system, RTG-D adds an additional communication time interval, \(T_{\text{comm}}\), to each \(c_k(t)\). If a section is successfully completed, but does not have enough time to arrive on the next destination node, it will be not only aborted in the end, but also will waste processor bandwidth which could otherwise be used for other (potentially feasible) thread sections. RTG-D avoids this by also checking whether each section has sufficient time for gossiping. (If a section is the last section of its thread, then there is no need to consider its feasibility for gossiping. Thus, \(T_{\text{comm}} = 0\).)

Suppose there are \(n\) thread 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: \(\rho_R(t) = \max \rho_{R_i}(t), \forall i\).

### C.3 Selecting Least-Local-Slack-First Section

Local section scheduling must consider not only meeting local section termination times, but also the time needed for the subsequent gossip. While DASA selects the section with the earliest (local) termination time in the feasible schedule for execution, RTG-D selects that with the least (local) slack. Through this way, RTG-D ensures that sections with less slack execute earlier and thus have more gossip time.

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 each section needs 0.5 time units, 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 another node.

RTG-D avoids this by following the least-slack-first order. In Fig. 4, we observe that 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 affect its gossip time. Note that RTG-D gains almost the same total slack time in these five sections as DASA does, but it allocates slack time more evenly, and thus 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 \(LS_i > LS_j\), and therefore \(j\) may be arranged before \(i\) in the schedule queue. To resolve this conflict, RTG-D “tightens” section \(i\)’s local slack time to the same as section \(j\)’s.

RTG-D’s schedule construction is described in Algorithm 1.

### C.4 Utility Propagation for Reducing Blocking Times

Thread section \(i\) may depend on section \(j\) on the same node, or depend on \(j\) on a different node. For the latter case, RTG-D propagates \(i\)’s utility to \(j\) in order to speed up \(j\)’s execution, and thus shorten \(i\)’s waiting time for the blocked resource. The utility propagation is done by gossiping to all system members within a limited time interval as it does in the head searching process.
Algorithm 1: RTG-D’s Section Scheduling Algorithm

1. Create an empty schedule \( \phi \).
2. For each section \( i \) in the local ready queue do
   1. Compute \( \text{Dep}(i) \), detecting and resolving deadlocks if any;
   2. Compute \( \text{PUD}_i \);
3. Sort sections in ready queue according to PUDs;
4. For each section \( i \) in decreasing PUD order do
   1. \( \hat{\phi} = \phi; \) /* get a copy for tentative changes */
   2. If \( i \not\in \phi \) then
      1. \( \text{CurrentLST} = \text{LST}(i); \) /* LST(\( i \)) returns the local slack of \( i \) */
      2. For each \( \text{PrevS} \) in \( \text{Dep}(i) \) do
         1. If \( \text{LST(PrevS)} \leq \text{CurrentLST} \) then
            1. Continue;
         2. Else
            1. \( \text{LST(PrevS)} = \text{CurrentLST}; \)
            2. \( \text{Remove(PrevS, } \hat{\phi}, \text{LST)}; \) /* Removes \( \text{PrevS} \) from \( \hat{\phi} \) at position LST */
            3. Insert(\( \text{PrevS}, \hat{\phi}, \text{CurrentLST} \));
      3. If \( \text{Feasible}(\hat{\phi}) \) then
         1. Insert(\( i, \hat{\phi}, \text{CurrentLST} \));
   5. If \( \text{Feasible}(\hat{\phi}) \) then
      1. \( \phi = \hat{\phi}; \)
6. Select least-slack section from \( \sigma \) for execution;

When \( j \)'s head node receives \( i \)'s gossip message, it will decide whether to continue \( j \)'s execution (thereby allowing \( j \) to return to \( i \)'s node and release the lock in the future), or immediately abort \( j \) and grant the lock to \( i \). (If a thread desires not to be aborted, then it must indicate so by setting an abort time of \( \infty \).) This decision is based on Global Utility Density (or GUD), which is defined as the ratio of the thread utility to the total remaining thread execution time.

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) 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.

C.5 Distributed Deadlock Resolution

Detecting deadlocks between different nodes require all system members to uniformly identify each thread. Thus, a global ID (or GID) is needed when a thread is first created. RTG-D regards determining GID as the “invocation zero” of a thread, and gossips to all system members within a limited time interval as it does in the head searching process. Each gossip message contains a tentative GID given by the node initiating the gossip, the thread GUD, and the thread’s arrival time. Ties in the initially proposed GID are broken by thread arrival times, and ties in arrival times are broken by GUD.

With GIDs, it is easier to determine the thread that must be aborted to resolve a distributed deadlock. Algorithm 3 describes this procedure.
Algorithm 2: RTG-D’s Utility Propagation Algorithm

Upon receiving a UP gossip message \(msg\):

1. COPY(gossip, msg);
2. if \(GUD_i > GUD_j\) then
3. \hspace{1em} if \(abt_j > cr_j\) then
4. \hspace{2em} abort \(j\);
5. \hspace{2em} gossip.lsr ← msg.lsr − abt_j;
6. \hspace{2em} give resource lock to \(i\);
7. \hspace{1em} else
8. \hspace{2em} continue \(j\)’s execution;
9. \hspace{2em} keep resource lock;
10. \hspace{1em} else
11. \hspace{2em} gossip.lsr ← msg.lsr;
12. \hspace{1em} gossip.round ← gossip.lsr / \(\Psi\);
13. \hspace{1em} gossip.c ← FANOUT(gossip.round);
14. \hspace{1em} RTG_GOSSIP(gossip);

Algorithm 3: RTG-D’s Distributed Deadlock Resolution Algorithm

Upon \(j\) receiving \(i\)’s UP gossip message \(msg\):

1. COPY(gossip, msg);
2. if \(DETECT(msg) = true\) then
3. \hspace{1em} if \(GUD_i > GUD_j\) then
4. \hspace{2em} abort \(j\);
5. \hspace{2em} gossip.lsr ← msg.lsr − abt_j;
6. \hspace{2em} give resource lock to \(i\);
7. \hspace{1em} else
8. \hspace{2em} continue \(j\)’s execution;
9. \hspace{2em} keep resource lock;
10. \hspace{1em} else
11. \hspace{2em} gossip.lsr ← msg.lsr;
12. \hspace{2em} gossip.round ← gossip.lsr / \(\Psi\);
13. \hspace{2em} gossip.c ← FANOUT(gossip.round);
14. \hspace{2em} RTG_GOSSIP(gossip);

IV. Algorithm Analysis

Let \(\delta\) be the desired probability for delivering a message to its destination node within the gossip period \(\Psi\). If the communication delay follows a Gamma distribution with a probability density function:

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

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:

\[
p = \left(\frac{C}{n - 1}\right) (1 - \sigma) (1 - \omega) \delta \tag{2}\]
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 $j$:

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

(3)

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

$$P[I(0) = j] = \begin{cases} 1 & j = 1 \\ 0 & j > 1 \end{cases} \quad P[I(t + 1) = j] = \sum_{i \leq j} P[I(t) = i] p_{i,j}$$

(4)

**Theorem 1:** 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 $t_{\text{max}}$ is given by:

$$p_{M_k} = \left\{ \begin{array}{ll} 1 - P[I(t_{\text{max}}) = \eta] & \times \frac{1}{U(t_{\text{max}})} = \left\{ 1 - \sum_{i \leq \eta} P[I(t_{\text{max}} - 1) = i] p_{i,\eta} \right\} \times \frac{1}{U(t_{\text{max}})} \end{array} \right.$$  

(5)

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

Let $w_k$ be the waiting time before section $k$’s execution. Now, $X_k$ and $X_m$ can be defined as:

$$X_k = \begin{cases} 1 & \text{If } w_k \leq LS_{rk} - LCD \\ 0 & \text{Otherwise} \end{cases} \quad X_m = \begin{cases} 1 & \text{If } w_k \leq LS_{rm} \\ 0 & \text{Otherwise} \end{cases}$$

(6)

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:

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

(7)

We got Theorem 2 and Lemma 3 in [8]:

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

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

(8)

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

**Theorem 4:** If a distributable 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 2, it will take section $i$ at most $t_{n_i}$ time rounds to gossip a VP message to section $j$’s head node.
After j’s head node receives i’s UP message, RTG-D will compare i’s GUD with j’s. If GUDi > GUDj, then j should give the lock to i as soon as possible. According to Algorithm 2, the handler will deal with j’s head within min(abtj, erj). According to Lemma 3, i’s head will expect a reply from j after at most tni time rounds. If tni − min(abtj, erj) ≥ LCD, then j can reply and give resource lock to i at the same time. Thus, i’s blocking time bound b1, j = 2tni. 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 LSri − LCD. Thus, if (LSri − LCD) − tni − min(abtj, erj) ≥ LCD, b1, j = LSri − 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 b1, j = 2tni, since j need not respond any more after the first reply to i. If GUDi ≤ GUDj, then j will not give i the resource lock until it finishes necessary execution. Thus, b1, j = LSri − LCD.

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

**Theorem 5:** RTG-D 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.

Fig. 5. Example Distributed Deadlock

Suppose GUDi > GUDj. Under the first condition, i will check the necessary time for deadlock solution, which is denoted as dsj2. Let LSri2 be the remaining local slack time of section i on Node 2, tni2 be the time rounds needed by i to gossip to Node 1 in order to finish i on time, and abtj1, abtj2 be the needed abortion time of section j on Node 1 and 2, respectively.

\[
dsj2 = \begin{cases} 
abtj2 & \text{No LIFO-ordered abortion is necessary from Node 1 to Node 2} \\
abtj1 + abtj2 + 2tni2 & \text{Otherwise} 
\end{cases} 
\]  

(9)

By LIFO-ordered abortion, we mean that 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.

\[
dsj2 = \begin{cases} 
tnj1 & \text{t}nj1 - abtj2 \geq LCD, \text{ or } tnj1 - (abtj1 + abtj2 + 2tni2) \geq LCD \\
\max (tnj1 + abtj1 + tni1, abtj2) & \text{Otherwise} 
\end{cases} 
\]  

(10)

Thus, if dsj2 ≤ LSri2 − LCD, the scheduler will resume i. Otherwise, it’ll abort i since i won’t have necessary remaining local slack time for gossiping.

The analysis is similar if GUDi > GUDj. The probability of blocking time bound is induced by RTG-D’s gossiping. It can be computed using (4), and a desired probability can be obtained by adjusting fan out C.

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

**Proof:** From Theorem 6 in [8], we directly obtain the failure exception notification time fn:

\[
fn = \begin{cases} 
3tn & \text{No LIFO-ordered abortion is necessary from Node m to Node n} \\
3tn + \sum _{i=m,...,n} tni & \text{Otherwise} 
\end{cases} 
\]  

(11)
V. Simulation Studies

We evaluate RTG-D’s effectiveness by comparing it with “RTG-D/DASA” — i.e., RTG-D with DASA as the section scheduler — as a baseline. We use uniform distribution to describe the 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 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.

Note that RTG-D 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-D/DASA does not use any of these techniques, but only follows RTG-D in the gossip-based searching of next destination nodes.

Fig. 6 shows the results for the eight-resource system. From the figure, we observe that RTG-D gives much better performance than RTG-D/DASA. Further, when OL is increased, both algorithms’ AUR and TMR decrease. We observed consistent results for the four-resource case, but those are omitted here for brevity.

In Fig. 7, as discussed in Section III, we observe that under any OL, RTG-D has a smaller variance of remaining local slack time than RTG-D/DASA, because it first executes sections with least local slack time instead of sections with earliest LTT. By this way, though sections’ mean value of remaining local slack time after execution is almost the same, RTG-D gives sections with less local slack time more chances to finish their gossip process. Again, we observed consistent results for the four-resource case.
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, (c) speeding-up the execution of blocking threads, and (d) detecting and resolving deadlocks. 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, deadlock detection and notification times, and failure-exception notification times for aborting partially executed sections of failed threads. Our simulation studies validate RTG-D's effectiveness.

Some example directions for extending our work include allowing multiple nodes to host same thread sections, unknown number of thread invocations, and non-step TUFs.

References