RTG-L: Dependably Scheduling Real-Time Distributable Threads in Large-Scale, Unreliable 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 real-time distributable threads in the presence of node/link failures and message losses in large-scale network systems. We present a distributed scheduling algorithm called RTG-L. The algorithm uses gossip-based communication for dynamically and dependably discovering eligible nodes. Traditionally, gossip protocols incur high message overhead. We explain that this problem is not that serious. We present a gossip-based message propagation protocol with lower message overhead. In scheduling local thread sections, RTG-L exploits slacks to optimize gossip time utilization. Thereby, it satisfies end-to-end time constraints with probabilistic assurance. Our simulation studies verify our analytical results.

I. Introduction

Many applications in distributed real-time systems are naturally structured as sequential execution flows, within or among objects, asynchronously or concurrently. The *distributable thread* programming model, which is supported in Sun's upcoming Distributed Real-Time Specification for Java (DRTSJ) [1], directly supports such execution flows as its first-class abstraction.

A distributable thread is a single control flow movement with logical distinction(i.e., having a globally unique identifier), extending and retracting through local and/or remote objects. In the rest of this 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), identity, and security credentials. The propagated thread context is used by real-time schedulers for resolving node-local resource contentions(e.g., CPU, I/O or lock contention), and for providing acceptably system-wide timeliness. Thus, threads constitute an abstraction for real-time scheduling. Figure 1 shows the execution of three threads [2].

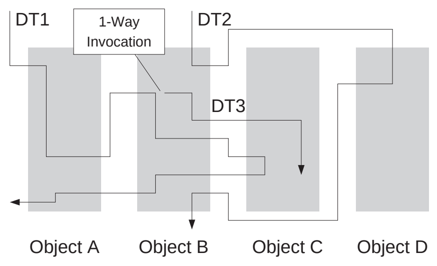


Fig. 1. Three Distributable Threads

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; and there are end-to-end properties that must be maintained, including timeliness, thread fault management, and thread control (e.g., concurrency, pause/resume, signaling of state changes).

In terms of providing direct support for causal sequential behaviors, threads can be viewed as at a higher level of abstraction than models such as Publish/Subscribe (P/S) [3]. 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 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 causal chains — programmed using threads or P/S — is conceptually similar.

We consider threads as the programming and scheduling abstraction in unreliable networks (e.g., those without a fixed network infrastructure, including mobile and wireless networks [4]), in the presence of application- and network-induced uncertainties. The uncertainties include 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 strong assurances on end-to-end thread timeliness behavior. Probabilistic timing assurances are often appropriate.

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. 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/rolling-back resources held by them to safe states (under a termination model). Untimely failure notifications can again be antagonistic to timeliness optimization—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 with Low Message Overhead* (or RTG-L), which provides assurances on thread time constraint satisfactions in large-scale unreliable networks. At its core, RTG-L contains a gossip protocol (e.g., see [6] and references therein). The algorithm uses this communication paradigm 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 boosts the time available for gossiping.

Traditionally, gossip protocols incur high message overheads. We explain that this problem is not that serious. We introduce a new message propagation protocol with low message overhead. Based on this work, we present RTG-L to schedule threads in unreliable large scale systems. Our simulation studies verify our analytical results.

Our work builds upon our prior work in [7] that presents the RTG-DS algorithm. Though RTG-L is also a gossip-based algorithm, it differentiates itself from RTG-DS by redesigning its core component — a message propagation protocol, in order to achieve fast propagation with less message overhead. End-to-end real-time scheduling has been studied in the past (e.g., [8]–[11]), but these are mostly limited to fixed infrastructure networks. End-to-end timing assurances in unreliable networks are considered in [7], [12], [13], but they do not deal with message overhead problems, which is precisely what our work does.

The rest of the paper is organized as follows: In Section II, we discuss models and algorithm objectives. Section III illustrates our gossip-based thread scheduling strategies. We present RTG-L algorithm in Section IV. In Section VI, we report our simulation studies. We conclude the paper and identify future work in Section VII.

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 section*. Thus, a thread can be viewed as being composed of a concatenation of thread sections.

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

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 $\mathbf{T} = \{T_1, T_2, T_3, \ldots\}$.

B. Timeliness Model and Utility Accrual Scheduling

Each thread's time constraint is specified using a time/utility function (or TUF) [14]. 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.

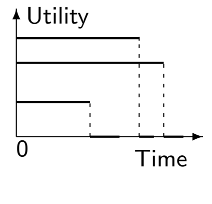


Fig. 2. Step TUFs

A thread T_i's TUF is denoted as $U_i\left(t\right)$. 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 $U_i\left(\right)$, 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. $U_i\left(t\right) > 0, \forall t \in \left[I_i, X_i\right]$ and $U_i\left(t\right) = 0, \forall t \notin \left[I_i, X_i\right], \forall i$.

If a thread has not completed by its termination time, 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. System Model

The network consists of a set of nodes, denoted $N = \{n_1, n_2, n_3, \ldots\}$, communicating through bidirectional links. A basic unicast routing protocol such as DSR [15] 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 [16].

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 [17]. Nodes may fail by crashing, links may fail transiently or permanently, and messages may be lost, all arbitrarily.

D. Objectives

Our goal is to design an algorithm that can schedule threads with probabilistic termination-time satisfactions in the presence of message losses and node/link failures —i.e., establish probabilistically satisfied end-to-end timing assurance for a thread. At the same time, we seek to reduce the message overhead during the scheduling process as much as possible. Further, we desire to maximize total thread accrued utility, and minimize the number of aborted threads.

III. Gossip-Based Thread Scheduling

Once a thread completes its execution on a node, referred to as the thread's *current* head node, that node determines the thread's *next* head node—i.e., the node on which the thread will make its next remote invocation, or return to, from its current invocation—to continue thread execution.

Since the next head node may crash, or may be unreachable, or may depart from the network, the current head node must dynamically discover the next head node, and determine whether or not the next head node can feasibly execute the thread section. This discovery is done using a gossip-style protocol, where the current head node randomly selects a set of peer nodes and multicasts the thread's identity and scheduling parameters, for a finite number of synchronous gossip rounds. Upon receiving the gossip message, a peer node repeats the gossip process, if it is not the next head node. If it is the next head node, the node determines the feasibility of executing the next thread section and replies back to the current head node with the result.

The current head node waits for a decision from its successor head node, but adds a deadline on this waiting time interval. If it does not receive a decision within that deadline, it will consider the next head node as crashed or unreachable. The current head node will immediately release the exception handler of the thread section for aborting the section, and will inform all previous thread head nodes regarding the thread abortion through a gossip process (so that they can release handlers).

The key aspect in determining the next head node in large unreliable networks, is to realize reliable and real-time (gossip) message propagation. As the network size increases, frequently keeping track of routes or link states will be less possible, and message losses will occur frequently and arbitrarily due to the inherent unreliability of the untethered network infrastructure. Since node crash failures are also assumed to occur arbitrarily, direct node-to-node communication would be inefficient, unless nodes monitor each other very frequently, which would be too expensive for network communication.

As previously mentioned, gossip-based protocols offer a scalable, robust, fault-tolerant, and probabilistically reliable message propagation design paradigm for large-scale, unreliable systems. In order to propagate a message in a system, informed nodes simply "gossip" to randomly selected targets, without requiring any confirmation from those targets regarding message reception.

Gossip-based algorithms incur relatively high message overheads. However, for message propagation in large-scale, unreliable networks, gossip does have attractive features. First, the random nature of gossip reduces the (lower-layer) overhead for gathering, storing, and updating massive amounts of information in a vast network—e.g., nodes update their link state tables less frequently. Second, gossip reliably spreads information all over the system (with computable probability), which helps head nodes determine their unknown successors, and propagates successors' information to potential users, despite node failures and message losses. Third, gossip is robust against a set of Byzantine attacks (e.g., blackhole attacks [18]).

IV. The RTG-L Algorithm

In RTG-L, nodes gossip when they need to determine their successor nodes. Also, they gossip when message loss ratio in the system is high. RTG-L makes nodes directly communicate with each other if message loss ratio is relatively low. However, the former condition is common in unreliable networks; thus we focus on gossip-based RTG-L in the paper.

A. Building Local TUF

RTG-L 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 $Er_{i,j}$. RTG-L 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-L 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-L determines the next node for executing the thread's next section, through a set of gossip rounds. The network communication delay incurred by RTG-L 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 by RTG-L to test for schedule feasibility, while constructing local thread section schedules.

B. Constructing Local Schedule

RTG-L constructs local schedules of thread sections, with the goal of maximizing the total accrued utility, maximizing the number of local termination times that are violated, and increasing the likelihood for the global thread termination time to be satisfied. Note that maximizing the number of satisfied local termination times contributes to increasing the likelihood for meeting global termination times, but that by itself is not sufficient. This is because of the communication delay incurred by RTG-L. While a section can be scheduled at its latest start time to complete before its local termination time, this can potentially waste the section's available local slack time, thereby prolonging the section's completion, and decreasing the available gossiping time for subsequent sections and thus the likelihood for the entire thread to complete before its global termination time. The problem of maximizing total accrued utility itself is NP-hard [19]. Thus, RTG-L considers two heuristics in constructing local section schedules: 1) Potential Utility Density (or PUD) and 2) remaining local slack time. A section's PUD is the utility that can be accrued by executing the section, per unit of execution time. For a section S_i, at a scheduling event that occurs at time t, its PUD is given by $PUD_i(t) = U_i(t + LEr_i(t)) = LEri(t)$, where $LEr_i(t)$ is S_i's remaining (local) execution time at t. Thus, a section's PUD measures its return on "investment". RTG-L constructs local section schedules at two scheduling events: 1) the arrival of a message that signals the release of the section for execution on the node; and 2) completion of the section's execution.

RTG-L constructs local schedules as follows. The algorithm sorts all sections in the ready queue in the descending order of their PUDs. The sorted sections are then examined, highest PUD first, and inserted into a tentative schedule. The tentative schedule is sorted in the ascending order of the section termination times, to minimize termination time misses (since deadline ordering is optimal for that objective), and tested for feasibility. A schedule is said to be feasible, if the predicted completion time of each section in the schedule does not exceed its local termination time. (This feasibility testing is similar to that in [19].) If the schedule is infeasible, the section is removed from the schedule. The process is repeated until all sections in the ready queue are examined, while preserving the invariant of schedule feasibility. The section with the least slack in the resulting schedule is then selected for execution, thereby allowing greater gossiping time for determining the thread's next node. The algorithm is shown in Algorithm 1.

Algorithm 1: Local RTG-L Scheduling Algorithm [Local_SCHEDULE( )]

```
1 Create an empty schedule $\phi$;
2 Let $t$ be the time of the scheduling event;
3 Sort sections in ready queue according to PUDs;
4 for each section in decreasing PUD order do
5     Insert section in $\phi$ at its termination time position (maintaining $\phi$'s increasing termination-time order);
6     if schedule is infeasible then
7         Remove section from $\phi$;
8 Select least-slack section from $\sigma$ for execution;
```

C. Discovering the Next Head Node for Thread Execution

We first introduce a gossip protocol with optimal message overhead, then describe the gossip-based RTG-L algorithm.

1) A Gossip Protocol with Low Message-Overhead: We describe the protocol by first introducing the necessary definitions.

Definition 1. *Gossip Round r: Denotes the r^{th} gossip time interval, at the beginning of which nodes send out messages. All messages are considered to arrive at their destination nodes when the round r ends.*

We assume that the message delay follows a non-negative distribution, e.g., the Gamma distribution [17]. Many distributions have infinite tails, and therefore, to determine the length of a gossip round, application users need to decide a termination time point t_{end}, after which message arrivals can be ignored. This is done by determining a threshold on the message arrival ratio, which is referred to as Θ. For instance, if $\Theta = 98\%$, we can determine the relative t_{end} in a given distribution. The length of a gossip round is then equal to the time interval between the round start time point (the value is often 0) and t_{end} in the distribution function.

Definition 2. I_r : *Denotes the number of newly informed nodes during gossip round* r*.*

Definition 3. U_r : *Denotes the number of uninformed nodes at the end of gossip round* r*.*

Definition 4. F_r : *The number of messages a node sends out at the beginning of gossip round* r*.*

At the end of gossip round r, the possibility that an uninformed node does not receive a message from a certain informed node, η, is

$$\eta = 1 - \frac{F_r}{N-1} \tag{1}$$

where N denotes the total number of nodes in the system. As a way similar to [17], we compute the expected number of uninformed nodes at the end of gossip round r:

$$U_r = U_{r-1} \times \eta^{I_{r-1}} = U_{r-1} \times (1 - \frac{F_r}{N-1})^{I_{r-1}} \tag{2}$$

When $F_r \ll N-1$, we have:

$$U_r = U_{r-1} \times \exp{(\frac{-F_r \times I_{r-1}}{N-1})} \tag{3}$$

The fan out and the number of messages issued during gossip round r (M_r), are shown in Equations 4 and 5, respectively:

$$F_r = \frac{N-1}{I_{r-1}} \times \ln(\frac{U_{r-1}}{U_r}) \tag{4}$$

$$M_r = F_r \times I_{r-1} = (N-1) \times \ln(\frac{U_{r-1}}{U_r}) \tag{5}$$

Different from gossip protocols with fixed fan out number at each round, here, F_r can be adjusted by application users.

Gossip protocols are fault-tolerant, but they have high message overheads — many informed nodes send messages to more target nodes — that may cause traffic congestion in the network.

The above gossip protocol can reduce the message overheads — it only lets the most recently informed nodes gossip once in the following round, instead of letting all informed nodes gossip repeatedly [7]. Besides, if I_r is large, the number of messages issued during round r can be adjusted by using Equations 4 and 5.

In gossip protocols, a message is supposed to be sent at the beginning of a round, and arrive at its destination before the end of the same round. This message should not be counted in the next round. Thus, the number of messages existing at the same time is much less than the total number.

In addition, randomly selecting gossip targets makes messages uniformly distributed in the network, thus, the likelihood of network congestion is reduced. Round Message Density at round r (RMD_r) is computed as following:

$$RMD_r = \frac{M_r}{N} \tag{6}$$

2) Protocol Description: When the message loss ratio is high, a head node utilizes gossip to determine the next head node. The informed next head node gossips its decision. Descriptions of the RTG-L on head nodes and intermediate nodes are shown in Algorithms 2, 3 and 4, respectively.

Algorithm 2: Gossip Protocol [GOSSIP( )]

```
1 On gossiping a message msg:
2 msg.r++ ;
3 Randomly selects msg.f targets ;
4 for each $i \in$ [1, ..., msg.f ] do
5     SEND ($target_i$, msg);
```

Algorithm 3: RTG-L on Head Node

```
1  Upon receiving a message msg:
2  accept = LOCAL_SCHEDULE(msg);
3  /* the node decides whether to provide the required service*/
4  msg.accept = accept;
5  GOSSIP(msg);
6  /*the node sends back its decision*/
7  if accept == TRUE then
8      EXECUTE();
9      BUILD(msg');
10     /* build new queries to determine the next head node */
11     WAIT(d);
12     /* wait till a designated deadline */
13     if msg'.accept == TRUE then
14         ABORT(holding section);
15         GOSSIP();
16         /* inform upstream former head nodes */
```

After sending out a query message to determine the next head node, a thread's current head waits for a reply till a certain deadline d (d should not be later than the thread's end-to-end termination time). If it does not receive any reply after this deadline, it will regard that the thread cannot be finished, abort the thread section, and uses gossip to inform the thread's upstream head nodes. For intermediate nodes, if a query message has been replied, they will not gossip it any more.

Algorithm 4: RTG-L on Intermediate Node

```
1 Upon receiving a message msg:
2 if (msg is a query message) then
3     if no reply yet then
4         GOSSIP(msg);
5 else
6     GOSSIP(msg);
```

V. Algorithm Analysis

1. RTG-L's Message Overhead Properties

Lemma 1. *The number of messages issued during all gossip rounds is* $\Theta(N \log N)$.

Proof: From Equation 5, we have

$$\Sigma_{r=1}^{r=R} M_r = (N-1) \times \Sigma_{r=1}^{r=R} \ln(\frac{U_{r-1}}{U_r}) \tag{7}$$

The number of issued messages during all gossip rounds is $\Theta(N \log N)$. ∎

According to [6], RTG-L's gossip has the optimal message overhead among all gossip protocols.

Theorem 2. *The number of issued messages is independent of* R, I_r, U_r *or* F_r.

Proof: The result is directly derived from Equation 5 and Lemma 1. ∎

2. RTG-L's Timeliness Properties

Theorem 3. *If all nodes in the system are underloaded, 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 determined by giving the expected number of informed nodes during gossip periods.*

Proof: Let p be the largest number of rounds a head node must wait for a reply. Let p_1 and p_2 be the number of rounds needed to determine the next head node and receive a decision. A section k's probability to satisfy its time constraint is:

$$p_{s_k} = \frac{I_{p_1}^T \times I_{p_2}^T}{N^2} \tag{8}$$

where $p_1 \geq 0$ and $p_2 \geq 0$, and $I_{p_1}^T$ and $I_{p_2}^T$ are total number of nodes during the current and next head node gossiping period, respectively. Thus, the probability P_{Sd} for a thread d to successfully complete its execution through $m+1$ head nodes, and that for a thread set D, P_{SD}, is given by:

$$P_{S_d} = \prod_{1\le k\le m} p_{s_k} \qquad\qquad P_{S_D} = \prod_{d\in D} P_{S_d} \tag{9}$$

■

To establish the algorithm's timeliness under overload condition, we first introduce the concept of UA scheduling and "best-effort" scheduling, then define the concept of Non Best-effort time Interval (or NBI) and Non Best-effort Ratio (or NBR).

When thread time constraints are expressed with Time Utility Functions (or TUFs, Section II-B), 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(see [7] for example algorithms).

UA algorithms that maximize total utility under downward step TUFs (Section II-B) default to EDF during underloads, since EDF satisfies all deadlines during underloads. Consequently, they obtain the optimum total utility during underloads. During overloads, they inherently favor more important threads over less important ones (since more utility can be attained from the former), irrespective of thread urgency, and thus exhibit adaptive behavior and graceful timeliness degradation. This behavior of UA algorithms is called "best-effort" in the sense that the algorithms strive their best to feasibly complete as many high importance threads — as specified by the application through TUFs — as possible. Consequently, high importance threads that arrive at any time always have a very high likelihood for feasible completion (irrespective of their urgency). Note also that EDFąŕs optimal timeliness behavior is a special-case of UA scheduling.

For reading convenience, we redefine Potential Utility Density (PUD) here. This term first appears in Section IV-B. Denote $U(T_i)$ the utility of a thread T_i. Denote $ET_rj(T_i)$ the remaining local execution time of T_i on node j. T_i's PUD on node j, $PUD_j(T_i)$, is:

$$PUD_j(T_i) = \frac{U(T_i)}{ET_rj(T_i)} \tag{10}$$

For convenience, we use "PUD" in the following descriptions.

Definition 5. *NBI : Consider a distributable thread scheduling algorithm A. Let a thread T_i be created at a node at a time t with the following properties: (a) T_i and all threads in A's execution-eligible thread set at time t are not feasible (system-wide) at t, but T_i is feasible just by itself; and (b) T_i has the highest PUD among all threads in A's execution-eligible thread set at time t.*

Now, A's NBI, denoted NBI_A, is defined as the duration of time that T_i will have to wait after t, before it is included in A's execution-eligible thread set. Thus, T_i is assumed to be feasible at $t+NBI_A$.

Note that RTG-L belongs to the independent node scheduling paradigm (i.e., it make its scheduling decisions using propagated thread scheduling parameters and without collaborating with other nodes).

Theorem 4. *RTG-L's worst-case NBI is δ, where δ is the algorithm scheduling overhead.*

Proof: RTG-L will examine T_i at t, since the arrival of a new thread is a scheduling event. Since T_i has the highest PUD and is feasible system-wide, it will include T_i's first section in its feasible (local) schedule at t, yielding a worst-case NBI of δ, the time constant involved for the algorithm to arrive at the local decision. ■

Now we establish the definitions of Non Best-effort Ratio (or NBR):

Definition 6. NBR : *Let a thread T_i be created at a node at a time t, under a certain distributable thread scheduling algorithm A. When T_i is created, there are a certain number of threads which have higher PUDs than T_i, disregarding whether the node is underloaded or overloaded.*

Denote $ET_t(T_i)$ the time duration of time that T_i will have to wait after t before its execution. Denote $ET_tmin(T_i)$ the minimum time duration among ET_t. Algorithm A's NBR for T_i, $NBR_A(T_i)$ is:

$$NBR_A(T_i) = MIN(\frac{ET_tmin(T_i)}{ET_t(T_i)}) \tag{11}$$

disregarding whether T_i is feasible or infeasible, and whether the node is underloaded or overloaded.

Theorem 5. *Disregarding the difference of scheduling overheads between different thread sets, a distributable thread scheduling algorithm A is a "best-effort" algorithm, if and only if its $NBR = 1$, for any thread T_i.*

Proof: As stated above, if algorithm A is "best-effort", it inherently favors more important threads over less important ones, irrespective of thread urgency. For a feasible thread T_i, the algorithm will schedule it according to its PUD, disregarding whether the node is overloaded or not. For an infeasible T_i, it is impossible to execute, and both $ET_t(T_i)$ and $ET_tmin(T_i)$ are ∞. Therefore, for a "best-effort" algorithm, its NBR is always 1.

Assume a "best-effort" algorithm B has $NBR_B(T_i) > 1$. That means for the thread T_i, there exists a time t, at which when T_i arrives, the algorithm will not schedule T_i according to its PUD — thus, the algorithm does not inherently favor more important threads over less important ones at any time. This contradicts the nature of "best-effort" algorithms. ■

Theorem 6. *RTG-L is a "best-effort" distributable thread scheduling algorithm.*

Proof: In local RTG-L scheduling, the algorithm always executes threads according to their relative PUD order (relative importance). Therefore, $NBR_RTG_L(T_i) = 1$ for any thread T_i. According

TABLE I

Gossip Protocol with Low Message Overhead

N	$I^T_{R,Thry}$	$I^T_{R,Sim}$	StdDev	Avg.RMD
300	299.5	298.62	0.75	0.95
400	399.5	399.58	0.67	1.03
500	499.5	499.54	0.69	1.19
600	599.5	598.58	0.66	1.07
700	699.5	699.38	0.73	1.01

to Theorem 5, RTG-L is a "best-effort" distributable thread scheduling algorithm. ■

From Theorem 6 we conclude that RTG-L first executes more important threads under overload condition, and thus possibly gain the maximum benefit from a certain thread set.

VI. Experimental Studies

In this section, we present simulation studies to evaluate RTG-L, including the gossip protocol with low message overhead, Local RTG-L algorithm, and finally, the RTG-L algorithm itself.

For the gossip protocol with low message overhead, we considered systems with different number of nodes. The pattern of I_r in each system at each round is arbitrarily selected. We obtained F_r from Equation 4, and computed average RMD using Equations 5 and 6. Note that F_r can be fractional. For instance, if $F_r = 2.5$, we use 2 and 3 interchangeably; thus over some gossip rounds, the expected F_r is 2.5. Table I compares theoretical results ($I^T_{R,Thry}$) with the simulation results ($I^T_{R,Sim}$), and presents the standard deviation (StdDev) and average round message densities (Avg.RMD) over 100 simulation experiments. From Table I, we observe that in large-scale systems (in Table I, each system contains 300-700 nodes), each $I^T_{R,Sim}$ complies well with its $I^T_{R,Thry}$ with a relatively small StdDev. Thus, the simulation results validate the analytical results presented in Section IV-C1. In addition, we also observe in Table I that the value of Avg.RMD is approximately 1 for all systems. This means that, on the average, a node sends out only 1 message during each gossip round, which is the designated maximum message propagation delay. Also, because nodes select gossip targets uniformly and randomly, the message overhead is uniformly distributed in the whole system. Thus, it is less possible for gossip to create traffic congestion in the network.

We evaluated the Local RTG-L algorithm (described in Section IV-B) by comparing it with the LSF and EDF algorithms. The simulation is also called "U/U Distribution" simulation, in order to feature thread sections that have interarrival times and execution times drawn from uniform probability distributions. In fact, these times always lie between zero and their designated maximum values. The

TABLE II

PROCESSOR LOAD

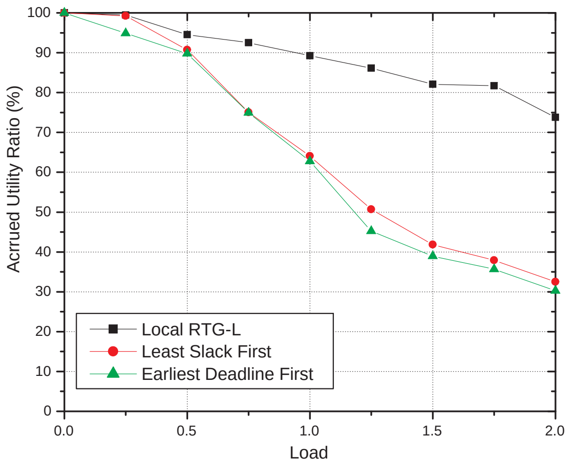
Max Interarrival Time (TUs)	**Expected Interarrival Time (TUs)**	**Processor Load**
400	200	0.25
200	100	0.5
133.2	66.6	0.75
100	50	1.0
80	40	1.25
66.6	33.3	1.5
50	25	2.0

maximum values are varied to examine algorithm behavior under different processor loads. All times in the following simulations are expressed in terms of Time Units (TUs).

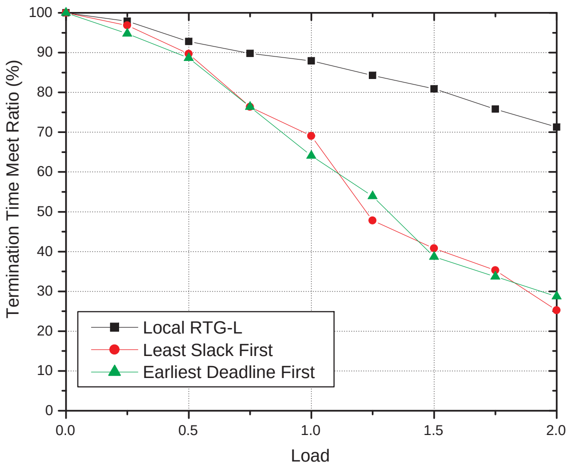
The termination times for each thread is also drawn from uniform probability distributions. The processor load metric (*Load*) is simply the expected time required to complete a thread section divided by the expected time between successive thread section arrivals. For the U/U Distribution simulations, the expected interarrival time is half of the maximum interarrival time. Similarly, the expected time remaining until a termination time when a thread section arrives is half of the maximum time remaining until that termination time. In this case, this is set to 100 TUs for our 100 created tasks. The required computation time for a given thread section is expected to be half of the time remaining until its termination time, or 50 TUs. By selecting maximum interarrival times from 800 to 50 TUs, the range of processor loads that can be examined extends from 0.25 to 2.0, respectively. The interarrival times and corresponding processor loads are shown in Table II.

Figures 3 shows the Accrued Utility Ratio (AUR) and the Termination time Meet Ratio (TMR) of the thread section set under increasing Message Loss Rate (MLR) and Intermediate Node Failure Rate (NFR) in a 700-node system, respectively. AUR is the ratio of the total accrued section utility to the maximum possible total section utility, and TMR is the ratio of the number of sections meeting their local termination times to the total number of thread sections on the node.

From Figures 3(a) and 3(b), we observe that Local RTG-L performs better than the other two algorithms. When $Load < 0.5$, the difference is not obvious — the workload is light, so each of the three algorithms can feasibly execute all existing sections. When $Load \geq 0.5$, the difference becomes much larger. When the workload increases, Local RTG-L drops sections with low PUD in favor of high-PUD sections, whereas LSF and EDF do not have a mechanism to check section feasibility and drop less important sections. Local RTG-L first executes high-PUD sections, thus, it can obtain more total utility from the same set of sections. Although using uniform distribution to model section



(a) AUR



(b) TMR

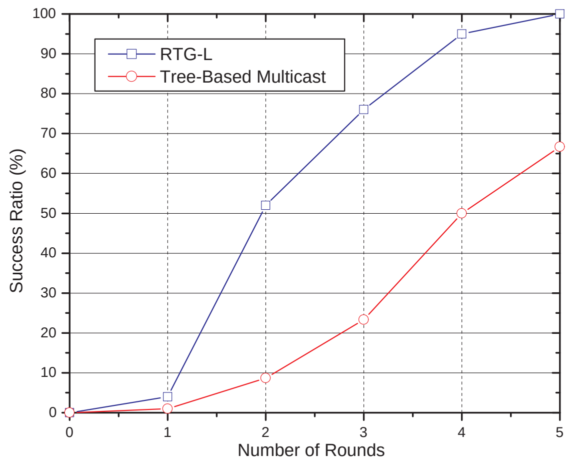
Fig. 3. AUR and TMR of Local RTG-L, LSF and EDF

behavior makes this gain not that obvious, Local RTG-L's AUR in Figure 3(a) still exceeds its TMR in Figure 3(b) under the same workload.

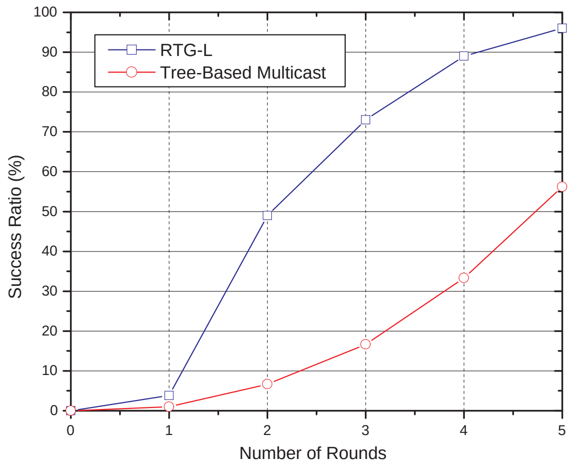
We evaluated the overall performance of the RTG-L algorithm in a 700-node unreliable system. We used a baseline algorithm called "Tree-Based Multicast" (or TBM), which propagates messages using a tree structure. The tree is constructed before message propagation, and the construction is based on the following rules:

1) Each parent node has at most 6 children;
2) To speed message propagation, each child node selects the highest-level available parent;
3) If there are more than one available parent at the same level, a child contacts the one with the shortest path.

We set the average message delay between a parent and its children as 0.2 gossip round, and the average time for a child to detect parent failure and find a new parent as 0.6 gossip round. Threads in the simulation environment executed through four nodes in a 700-node system, so it should make three remote invocations to find next head nodes. During any invocation, if a head node cannot timely receive a reply from the next head node, it will abort the thread section and announce the failure of the thread. Thus, a thread is successfully finished, only if all of its head nodes find their next head nodes on time. The designated time interval for a remote invocation is represented by the number of rounds. Success Ratio (SR) is the number of successfully executed sections over the total number of sections.



(a) Condition of Message Losses

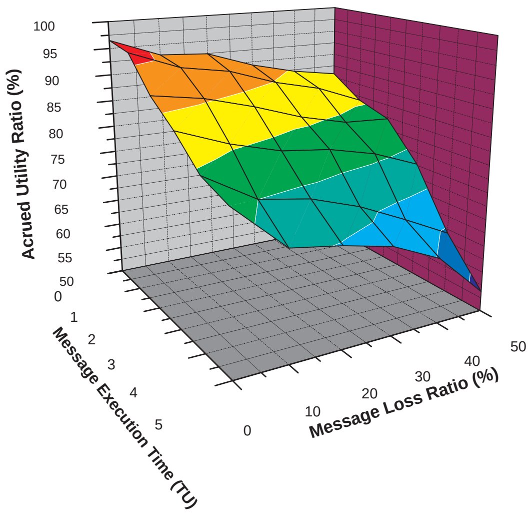


(b) Condition of Node Failures

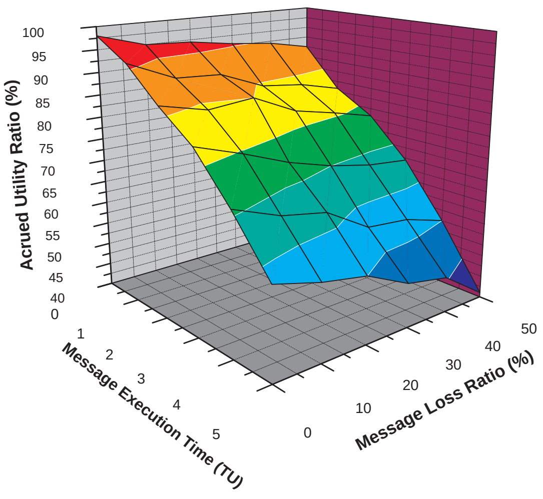
Fig. 4. SR of RTG-L and TBM

Figures 4(a) and 4(b) depict the SR of Local RTG-L and TBM under message losses and node failures, respectively. We set the message loss ratio and node failure ratio as 35%. From both figures, we observe that RTG-L performs much better than TBM. TBM depends on a fixed tree structure, thus, it cannot adjust the number of messages according to different time constraints. In addition, message losses and node failures happen frequently. If a message is lost during propagation, the parent has to wait for some time, and retransmit the same message if it does not receive a confirmation from a child. This not only delays the child, but also delays all nodes in that child's subtree. In both figures, if the time interval is only 1 round, in an unreliable environment, it is less possible to inform the next head node on time. Thus, both algorithms' have a low SR (RTG-L's SR is about 4%, while TBM's SR is about 1%). When the time interval becomes longer, both algorithms perform better, but RTG-L's SR increases much faster. If a node failure occurs, TBM's performance becomes worse — a child has to spend 0.6 gossip round to determine a new parent, and then wait for the message transmission (we observe this in Figure 4(b)). The performance of RTG-L also degrades, but not significantly. So the difference between these two algorithms is larger in Figure 4(b) than that in Figure 4(a).

We compared the RTG-L algorithm with a previously presented scheduling algorithm called RTG-DS [7], under the condition of different Message Execution Time (MET) and Message Loss Ratio (MLR), in a 700-node unreliable system. Here, MET is the time taken to process ONE received message or send ONE new message during a gossip round. Because message propagation is the key process in distributed real-time scheduling, message processing is always given the highest priority



(a) RTG-L



(b) RTG-DS

Fig. 5. AUR of RTG-L/RTG-DS

to execute. In the following simulation, MET varies from 0 to 5 TUs, while the designated max execution time of each thread section is 400 TUs. In addition, there are totally 50 thread sections in the simulated system, and they may gossip messages at the same time. Thus, the possible maximum message-processing overhead on a node is the sum of sending and receiving messages, that is, $100 * MET$ (a node sends messages at the beginning a gossip round, and receives messages during the whole round).

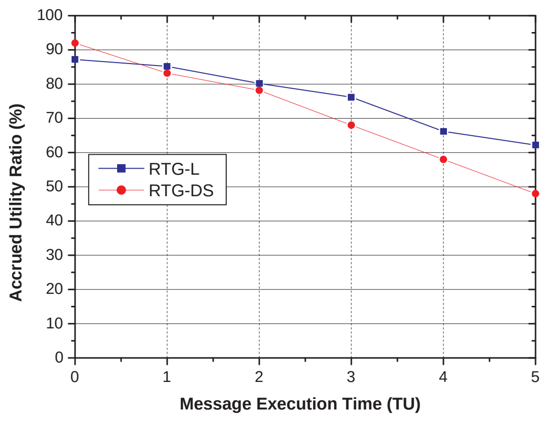
RTG-DS is very different from RTG-L, in the sense that in RTG-DS, the fan out number (F_r) at each gossip round is fixed, while in RTG-L F_r can be varied at different rounds. In addition, in RTG-DS, every "infected" node gossips till the end, while it only gossips once in RTG-L. Thus, RTG-L has a lower message overhead in communication processes.

In this simulation, as in the above Local RTG-L simulation, we used U/U distribution to model threads, and set a processor's thread section execution load $Load = 1$. If $MET = 0$, each node is probabilistically underload. Otherwise, a node has to process messages and drop thread sections.

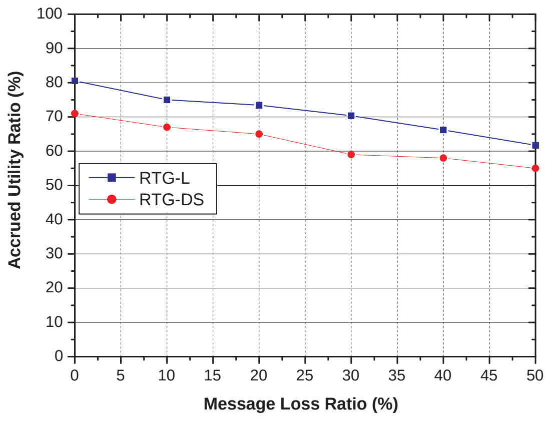
Figure 5(a) and 5(b) shows the AUR of RTG-L and RTG-DS, respectively. From both figures, we may observe that as MET and MLR increase, both AURs decrease. When MET increases, a node has to use more time to handle messages, and decrease the time for executing thread sections. When MLR increases, a node has less chance to receive messages for thread execution, thus AUR also decreases.

We can make a clearer observation in Figure 6 that shows cross section figures in Figure 5. In Figure 6(a) (MLR is fixed to 40%), we observe that when $MET = 0$, RTG-DS performs better. Under

RTG-DS, an "infected" node gossips at each round instead of gossiping only once. Without considering MET, messages containing scheduling information propagated more reliably under RTG-DS. However, when MET increases, RTG-L performs better, because nodes have lower message overhead, and have more time to execute thread sections. In Figure 6(b) (MET is fixed to 4 TUs), we observe that RTG-L always performs better. Since less received messages always means less message-processing overhead.



(a) AUR under different METs



(b) AUR under different MLRs

Fig. 6. AUR under different MET/MLR

VII. Conclusions and Future Work

In this paper, we present a gossip-based algorithm called RTG-L, for scheduling distributable threads in unreliable networks. RTG-L especially focuses on lowering message overhead in gossip, and uses low-overhead gossip protocol to propagate thread scheduling parameters, and determine successive nodes for feasible thread execution. In addition, it constructs local thread section schedules by exploiting thread slack in a way that enhances time available for message propagation. Our simulation studies validate the algorithm's effectiveness.

Immediate directions for extending our work are designing lower message overhead and faster gossip-based protocols, and allowing node anonymity, unknown number of thread sections, 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] 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.

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

[4] 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.

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

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

[7] K. Han et al., "Exploiting slack for scheduling dependent, distributable real-time threads in mobile ad hoc networks," in *International Conference on Real-Time and Network Systems (RTNS)*, March 2007.

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

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

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

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

[15] 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.

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

[17] S. Verma and W. Ooi, "Controlling gossip protocol infection pattern using adaptive fanout," in *The 25th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2005.

[18] B. Awerbuch, Reza Curtmola, et al., "Mitigating byzantine attacks in ad hoc wireless networks," Tech. Rep. I, JHU CS Dept., March 2004.

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