

# On Utility Accrual Real-Time Channel Establishment in Multi-hop Networks

Karthik Channakeshava and Binoy Ravindran

Real-Time Systems Laboratory

Bradley Department of Electrical and Computer Engineering

Virginia Tech, Blacksburg, VA 24061, USA

E-mail: {kchannak, binoy}@vt.edu

## Abstract

*We consider Real-Time CORBA 2.0 (Dynamic Scheduling) distributable threads operating in multi-hop networks. When distributable threads are subject to time/utility function-time constraints and timeliness optimality criteria such as maximizing accrued, system-wide utility, utility accrual real-time channels must be established. Such channels transport messages that are generated as distributable threads transcend nodes, in a way that maximizes system-wide, message-level utility. We present a utility accrual channel establishment algorithm called Local Decision for Utility accrual Channel Establishment (or LocDUCE). Since the channel establishment problem is NP-hard, LocDUCE heuristically computes channels. We implement the algorithm in a prototype test-bed and experimentally compare it with the Open Shortest Path First (OSPF) routing algorithm. Our experimental measurements reveal that LocDUCE accrues significantly higher utility than OSPF.*

## 1 Introduction

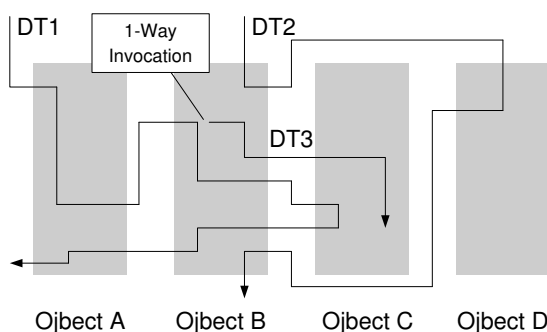
The Object Management Group's recently adopted Real-Time CORBA 2.0 (Dynamic Scheduling) standard [1] (abbreviated here as RTC2<sup>1</sup>) specifies *distributable threads* as a programming and scheduling abstraction for system-wide, end-to-end scheduling in real-time distributed sys-

---

<sup>1</sup>Real-Time CORBA 2.0 has been recently renamed as Real-Time CORBA 1.2.

tems. Distributable threads first appeared in the Alpha OS [2] and later in Alpha’s descendant, the MK7.3 OS [3,4].

A distributable thread is a single thread of execution with a globally unique identifier that transparently extends and retracts through an arbitrary number of local and remote objects. A distributable thread is thus an end-to-end control flow abstraction, with a logically distinct locus of control flow movement within/among objects and nodes. Concurrency is at the distributable thread-level. Thus, a distributable thread always has a single execution point that will execute at a node when it becomes “most eligible” as deemed by the node scheduler. In the rest of the paper, we will refer to distributable threads as *threads* except as necessary for clarity.



**Figure 1. Distributable Threads**

A thread carries its execution context as it transits node boundaries, including information such as the thread’s scheduling parameters (e.g., time constraints, execution time, importance), identity, and security credentials. Hence, threads require that Real-Time CORBA’s *Client Propagated* model be used, not the *Server Declared* model. Figure 1 cited from [1] shows the execution of threads.

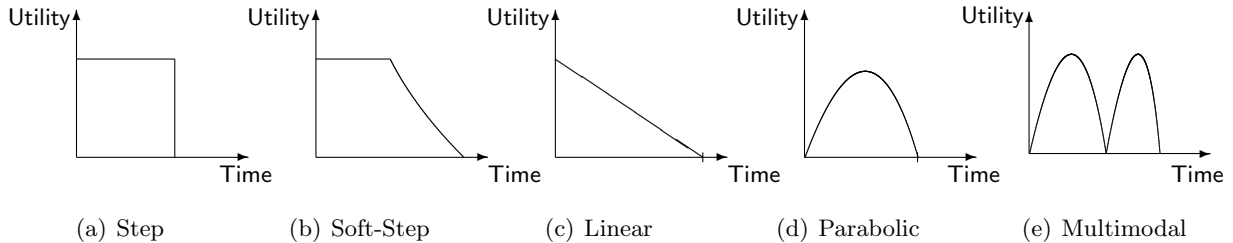
The propagated parameters are used by the schedulers on each of the nodes the thread transits, for resolving all node-local resource contentions among threads and for scheduling threads on nodes to satisfy the system’s timeliness optimality. Using the same optimality criterion, with the same parameters on each node that a thread transits, results in approximate, system-wide timeliness optimality. RTC2 explicitly supports this distributed scheduling approach, called *Distributed Scheduling: Case 2* in the RTC2 specification, due to its simplicity and capability for coherent end-to-end scheduling.<sup>2</sup>

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

## 1.1 TUFs and UA Scheduling

In this paper, we focus on complex, dynamic, adaptive real-time systems at any level(s) of an enterprise—e.g., in the defense domain, from devices such as multi-mode phased array radars [5] to battle management [6]. Such systems include “soft” as well as hard time constraints in the sense that completing a time-constrained activity at any time will result in some utility to the system, which depends on the activity’s completion time. Such soft real-time constraints may be as important or mission-critical as hard deadlines.

Jensen’s time/utility functions [7] (or TUFs) allow the semantics of soft time constraints to be precisely specified. A TUF specifies the utility to the system that results from the completion of an activity as a function of its completion time. Figure 2 shows the conventional deadline (downward step) and several soft time constraints specified using TUFs.



**Figure 2. Deadline and Example Soft Time Constraints Specified Using TUFs**

When time constraints are expressed with TUFs, the scheduling optimality criteria are based on factors that are in terms of maximizing accrued utility from those activities—e.g., maximizing the sum, or the expected sum, of the activities’ attained utilities. Such criteria are called Utility Accrual (UA) criteria, and sequencing (scheduling, dispatching) algorithms that consider UA criteria are called UA sequencing algorithms. In general, other factors may also be included in the optimality criteria, such as resource dependencies and precedence constraints. Several UA scheduling algorithms are presented in the literature [8–14]. RTC2 has IDL interfaces for the UA scheduling discipline, besides others such as fixed-priority, earliest deadline first, and least laxity first.

## 1.2 UA Channel Establishment

When a multi-hop network – i.e., one where end-hosts are interconnected by multiple switches and routers – is considered as the underlying platform for an RTC2 application, distributable threads will compete for node-local resources as well as network resources. Node-local resources

are those resources that are local to a system “node” such as an end-host or a router. Such resources include physical resources (e.g., processor, disk, I/O) and logical resources (e.g., locks).

Network resources include *real-time channels*. Traditionally, a real-time channel is a unidirectional virtual circuit that is established for application-level messages in a multi-hop network with guaranteed timeliness properties [15]. In the context of an RTC2 application, application-level messages include those that are generated when distributable threads invoke operations on remote objects and thus transcend nodes. Thus, such messages contend for real-time channels in multi-hop networks. Moreover, they are indirectly subject to the timeliness properties of their “parent” distributable threads, on which time constraints and timeliness optimality criteria are explicitly expressed.

While scheduling of threads on nodes and resolution of node-local resource contention among threads is performed by a scheduling algorithm, real-time channels are established by a channel establishment algorithm. Thus, when threads are subject to TUF time constraints and UA optimality criteria, UA scheduling algorithms and UA channel establishment algorithms must be used for coherent system-wide resource management and for improved timeliness optimization.

In this paper, we consider the problem of UA channel establishment in multi-hop networks. We consider thread time constraints that are specified using TUFs and the optimality criterion of maximizing the sum of threads’ attained utilities. We focus on messages of distributable threads. Thus, thread messages are indirectly subject to TUF time constraints. Toward maximizing the sum of threads’ attained utilities, we consider the problem of establishing channels for thread messages such that the sum of *messages’* attained utilities are maximized.

Note that timeliness optimization at the message-level is completely consistent with RTC2’s Case 2 approach, as thread messages, on behalf of threads, contend for real-time channel resources. This contention is simply resolved (for the messages) by using the thread scheduling parameters (that messages propagate) and by considering the thread-level optimality criterion at the message-level. Thus, message-level timeliness optimization contributes to thread-level timeliness optimization.

The UA channel establishment problem can be shown to be  $\mathcal{NP}$ -hard. Thus, we present an algorithm called Local Decision for Utility accrual Channel Establishment (or LocDUCE) that heuristically computes channels, seeking to maximize the sum of messages’ attained utilities as much as possible. We implement LocDUCE in a prototype test-bed and experimentally compare it with the Internet standard Open Shortest Path First (OSPF) routing algorithm. Our experimental measurements reveal that LocDUCE accrues significantly higher utility than

OSPF.

Thus, the contribution of the paper is the LocDUCE algorithm and its experimental validation. Most of the past efforts on real-time channel establishment [15–23], focus on the deadline time constraint. To the best of our knowledge, we are not aware of any other efforts that address the problem of UA channel establishment.

### 1.3 Organization of Paper

The rest of the paper is organized as follows: In Section 2, we provide motivation for the TUF/UA model by summarizing two significant applications that were successfully implemented using that model. We describe our message, timeliness, and system models in Section 3. Section 4 presents the LocDUCE algorithm and Section 5 discusses the algorithm implementation. We discuss the experimental evaluation of LocDUCE in Section 6. Finally, we conclude the paper in Section 7.

## 2 Motivating Application Examples

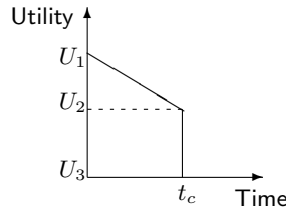
As example real-time systems requiring the expressiveness and adaptability of TUF time constraints, we summarize TUFs of two applications. These include: (1) AWACS (Airborne Warning and Control System) surveillance mode tracker system [24] built by The MITRE Corporation and The Open Group (TOG); and (2) a coastal air defense system [25] built by General Dynamics (GD) and Carnegie Mellon University (CMU). We only summarize some of the application time constraints here; other details can be found in [24, 25], respectively.

### 2.1 TUFs in AWACS

The AWACS is an airborne radar system with many missions, including air surveillance. Surveillance missions generate aircraft tracks for command and control (C2) and battle management (BM). The surveillance tracker consists of several different activities. Its most demanding computation, called *association*, associates sensor reports to aircraft tracks. The tracker employs two sensors that sweep 180 degrees out of phase with a ten second period. Thus, association has a “critical time” at the period length. If the computation can process a sensor report for a track in under five seconds (half the sweep), that will provide better data for the corresponding report from the out-of-phase sensor. Thus, prior to critical time, utility of association decreases as critical time nears.

After the critical time, the utility of association is zero, because newer sensor data has probably arrived. Thus, if the processing load in one sensor sweep period is so heavy that it cannot be completed, probably the load will be about the same in the next period. So there will not be any resources to also process sensor data from the previous sweep.

This timeliness behavior, which requires the expressiveness and adaptability of soft yet mission-critical time constraints, would be difficult to describe using priorities. An effective solution is to describe it using TUFs.



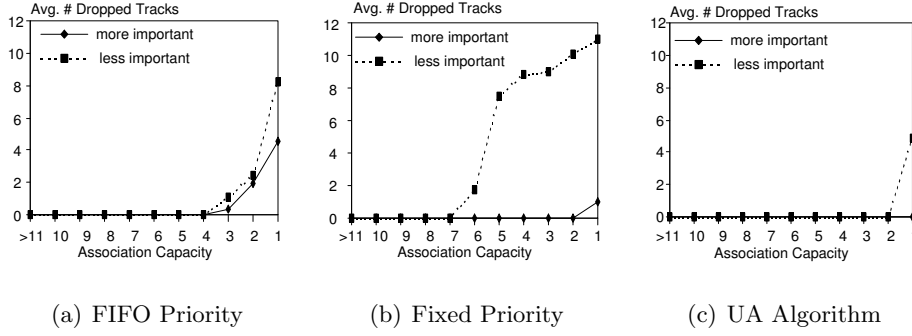
**Figure 3. AWACS Association TUF**

The described semantics establish association’s TUF shape: a critical time  $t_c$  at the sweep period; utility that decreases from a value  $U_1$  to a value  $U_2$  until  $t_c$ ; and an utility value  $U_3$  after  $t_c$ .  $U_1$ ,  $U_2$ , and  $U_3$  are determined using Application QoS (AQoS) metrics such as: (1) track quality, which is a measure of the amount of sensor data incorporated in a track record; (2) track accuracy, which is a measure of the uncertainty in the estimate of a track’s position and velocity; and (3) track importance, which is measure of track attributes such as its threat. Figure 3 shows the association thread’s TUF.

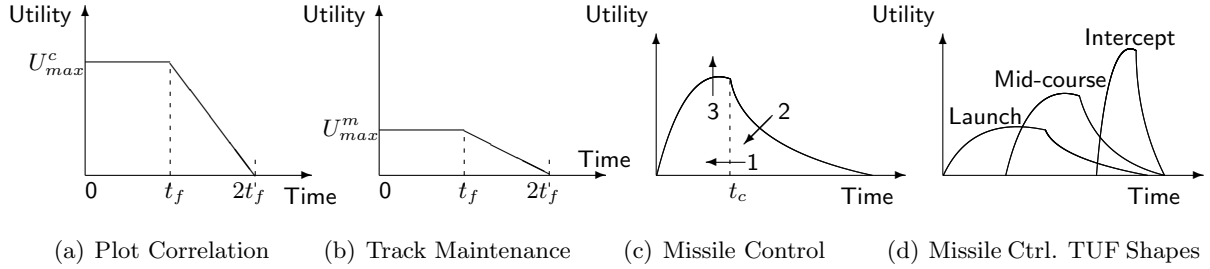
The tracker creates threads for each airborne object that it tracks. The threads perform a sequence of activities, including association. The TUFs of all threads have the same basic shape shown in Figure 3, but use different values for  $U_1$ ,  $U_2$ , and  $U_3$ . The system’s UA scheduling algorithm resolves the resource contention among all the association (and other) threads and schedules system resources to maximize the total summed utility.

The AWACS surveillance tracker implementation was done using TOG’s MK7 operating system [3]. MK7 contains the UA scheduling algorithm described in [9]. To understand how well MK7’s UA algorithm is able to schedule system resources in a mission-oriented way, significant performance measurements were made. Different scheduling algorithms, including FIFO and fixed priority, were compared with [9].

Figure 4 shows the average number of dropped tracks for the three scheduling policies under decreasing association capacity. The figure illustrates that the UA algorithm minimizes the number of dropped tracks, thereby illustrating the adaptivity of the TUF/UA paradigm.



**Figure 4. Avg. Dropped Tracks Under Decreasing Assocn. Capacity**



**Figure 5. TUFs of Three Activities in GD/CMU Coastal Air Defense**

Time constraints of three activities in the GD/CMU coastal air defense system, called *plot correlation*, *track maintenance*, and *missile control* are shown in Figures 5(a), 5(b), and 5(c), respectively. Figure 5(d) illustrates how the shape of the missile control activity's TUF dynamically changes. Details of how the TUFs were derived, application implementation, and adaptive timeliness measurements can be found in [25].

### 3 The Models

#### 3.1 Message Model

We consider inter-node messages that are generated when distributable threads invoke operations on remote objects (and thus transcend node boundaries). Thus, all messages are assumed to be created as a result of threads' remote operation invocations.

We denote the set of messages as  $m_i \in M, i \in [1, n]$ . The bit length of a message  $m_i$  at the data link-layer is denoted as  $b(m_i)$ . The physical framing overheads increase this size into an actual bit length  $b'(m_i) > b(m_i)$  for transmission. Thus, the transmission latency of a message  $m_i$  is given by  $l_i = b'(m_i)/\psi$ , where  $\psi$  denotes the nominal throughput of the underlying network, e.g.,  $10^9$  bits/s for Gigabit Ethernet.

We consider the *unimodal arbitrary arrival model* for messages, as this model “dominates” other arrival models including aperiodic, sporadic, and periodic arrival models due to the “strength” of the “adversary” embodied in the model [26]. For a message  $m_i$ , the unimodal arrival model defines the size of a sliding time window  $w(m_i)$  and the maximum number of arrivals  $a(m_i)$  that can occur during that window.

### 3.2 Timeliness Model

Each message  $m_i \in M$  has a time constraint that is expressed using a TUF. The TUF time constraint of a message is derived from the time constraint of the distributable thread to which the message belongs. We denote message  $m_i$ ’s TUF as  $U_i(\cdot)$ . Thus,  $m_i$ ’s arrival at its destination host application-layer (which triggers the invoked operation on an object on the host) at a time  $t$  will yield an utility  $U_i(t)$ .

Though TUFs can take arbitrary shapes, we restrict our focus to non-increasing, *unimodal* TUFs. Unimodal TUFs are those functions for which any decrease in utility cannot be followed by an increase [7]. Figures 2(a), 2(b), 2(c) and 2(d) show examples. TUFs that are not unimodal are called *multimodal*. Figure 2(e) shows an example.

Non-increasing unimodal TUFs are simply those unimodal TUFs for which utility never increases as time advances. Figures 2(a), 2(b), and 2(c) show examples. The class of such TUFs allow specifying a broad range of time constraints; hence we focus on them.

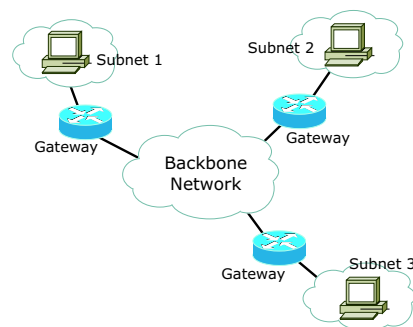
Each message  $m_i$ ’s TUF  $U_i$  has an initial time, denoted as  $I(m_i)$ , and a termination time, denoted as  $X(m_i)$ . Initial time is the earliest time, and termination time is the latest time, for which the TUF is defined i.e.,  $U_i(\cdot)$  is only defined in the interval  $[I(m_i), X(m_i)]$ . If the termination time is reached and the corresponding thread has not finished its execution, then an exception is raised. Usually, the exception causes abortion of the thread.

If the termination time is reached when a message of the thread is in transit on a channel toward the host where the remote object is located (on which an operation is invoked by the thread), the exception is raised at the next immediate node in the message’s channel. This node can either be the destination host or an intermediate node such as a router or a switch. In such a case, the exception is raised when the message arrives at the node and triggers the node scheduler. Typically, UA schedulers such as [14] “drops” such messages.

We assume that,  $U_i(t) \geq 0, \forall t \in [I(m_i), X(m_i)], i \in [1, n]$ .

### 3.3 System Model

We consider a multi-hop network with an arbitrary topology that consists of several networks of various types. A typical example is a local area network (or LAN) that interconnects a set of hosts using one or more switches. A collection of such LANs can further be interconnected together by a set of routers, thus forming a wide area network (or WAN). In such a network, a message can travel from a source host to a destination host by passing through a number of intermediate nodes (switches and routers). The route from a source to destination thus includes a set of nodes where a message can be stored and forwarded to the appropriate next hop. Figure 6 shows an example of such a network.



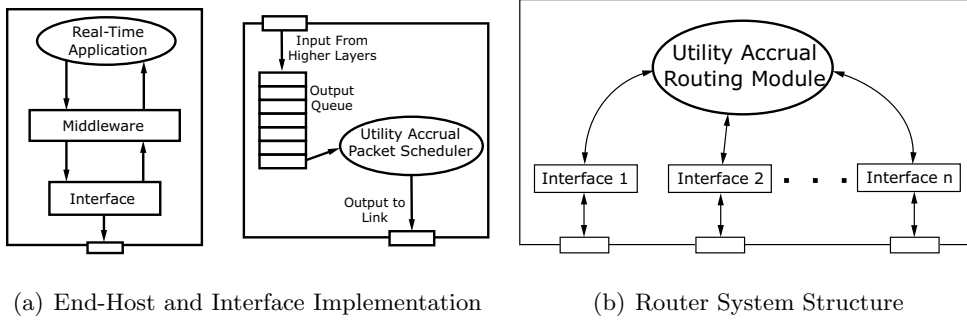
**Figure 6. A Multi-hop Network**

Figure 7 shows the components used in the system model. We assume that each system node – end-hosts, switches, and routers – are equipped with the UPA (utility accrual packet scheduling algorithm) presented in [14]. Figure 7(b) and 7(a) show the router and the end-host and interface implementations, respectively.

A message that is generated when a distributable thread invokes operation on a remote object flows through the middleware and operating system layers and gets decomposed into one or more message packets, when the message size exceeds a packet size. When message decomposition takes place, all packets of the message will inherit the scheduling parameters of the message.

The packets arrive at the MAC-layer and are placed on the output queue for the outgoing network link. When the link becomes physically free for transmission, the (UPA) packet scheduler selects a packet from the queue for transmission toward an intermediate node such as a router or gateway.

We assume that each router maintains a routing table, state information that includes distributable thread characteristics, and next-hop information. When a packet arrives at a router, it triggers the channel establishment algorithm. The algorithm computes the “next hop” for the



**Figure 7. System Node Components**

packet, selects the outgoing network link/interface, and places the packet on the corresponding output queue. As before, when the link becomes physically free, it triggers the packet scheduler, which then selects a packet from the queue for transmission.

In the rest of the paper, we will refer to packets as messages for simplicity, except as necessary for clarity i.e., each message is assumed to be transported as a single packet.

We assume that the clocks of all nodes are synchronized using a protocol such as [27]. The clock synchronization (or clock-sync, for short) module at each node periodically generates clock-sync packets at a period  $\theta$ . Clock-sync packets are always transmitted before application packets are transmitted. Bit length of a clock-sync packet at the data link layer is a constant and is denoted  $b_c$ . Physical framing overheads increase this to  $b'_c > b_c$ .

## 4 The LocDUCE Algorithm

We follow a “bottom-up” approach in describing LocDUCE. The key step in the algorithm is estimating the delay incurred by a message on a single network hop. Since we assume that message packets are scheduled by the UPA algorithm at all system nodes, we determine the single hop delay by analyzing delays under UPA.

Thus, we first overview the UPA algorithm. We then analyze the single hop delay under UPA. Thereafter, we present the LocDUCE algorithm.

### 4.1 Overview of UPA

UPA is a packet scheduling algorithm that executes at the MAC-layer of system nodes (e.g., hosts, switches) for selecting packets for outbound transmission. The algorithm considers a packet model, where packets have non-increasing, unimodal TUFs and seeks to maximize the sum of packets’ attained utilities.

UPA first constructs a tentative schedule by sorting packets in decreasing order of their “return of investments.” The return of investment for a packet is the potential timeliness utility that can be obtained by spending a unit amount of network transmission time for the packet. Thus, “high return” packets will appear early in the tentative schedule. The return of investment for a packet is determined by simply computing the ratio of the maximum possible packet utility (specified by the packet TUF) to the packet termination time. In [14], we call this ratio “pseudo-slope,” since it only gives an approximate measure of the TUF slope.

From this tentative schedule, packets that are found to be infeasible are moved to the end of the schedule. Infeasible packets are packets that cannot arrive at their destinations before their deadlines, no matter what. This is because, the *remaining* transmission time of such packets are longer than the time interval between their arrival at an end-host or the switch and the packet deadlines. Packets that are not infeasible are feasible packets.

To determine whether a packet is infeasible, the algorithm therefore needs global time. Thus, all system nodes are assumed to be equipped with clock synchronization modules and have access to synchronized clocks.

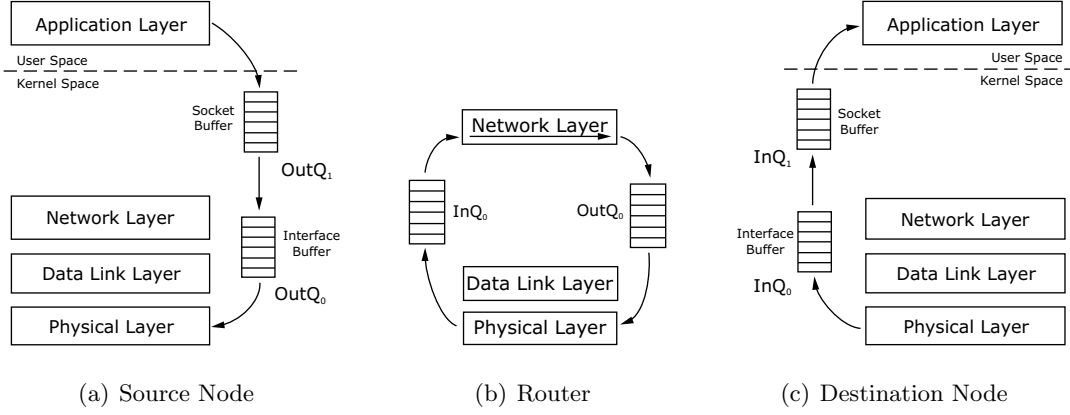
Once infeasible packets are moved to the schedule-end, the algorithm maximizes the *local* aggregate utility in the resulting schedule. The local aggregate utility is maximized by observing that given two schedules  $\sigma_a = \langle \sigma_1, p_i, p_j, \sigma_2 \rangle$  and  $\sigma_b = \langle \sigma_1, p_j, p_i, \sigma_2 \rangle$  of a packet set  $\mathcal{A}$ , such that  $\sigma_1 \neq \emptyset$ ,  $\sigma_2 \neq \emptyset$ ,  $\sigma_1 \cup \sigma_2 = \mathcal{A} - \{p_i, p_j\}$ , and  $\sigma_1 \cap \sigma_2 = \emptyset$ , the scheduling decision at a time  $t$ , where  $t = \sum_{k \in \sigma_1} l_k$ , that will lead to maximum local aggregate utility is determined by computing  $\Delta_{i,j}(t)$ , where  $\Delta_{i,j}(t) = [U_i(t + l_i) + U_j(t + l_i + l_j)] - [U_j(t + l_j) + U_i(t + l_j + l_i)]$ . Thus, if  $\Delta_{i,j}(t) \geq 0$ , then schedule  $\sigma_a$  will yield a higher aggregate utility than  $\sigma_b$ .

UPA maximizes local aggregate utility by examining adjacent pairs of packets in the schedule, computing  $\Delta$ , and swapping the packets, if the reverse order can lead to higher local aggregate utility. The procedure is repeated until no swaps are required. The packet that appears first in the resulting schedule is then selected for transmission.

## 4.2 Single Hop Delay

Figure 8 shows the typical set of input and output queues through which message packets flow from the application-layer in a source node, via an intermediate-node such as a router, to the application-layer in a destination node. To estimate the end-to-end delay incurred by a message packet from the source application-layer to the destination application-layer, the total delay incurred by the packet in all the queues must be accounted for. Note that in our system

model, all packet queues are scheduled by UPA.



**Figure 8. Node Input/Output Queues**

#### 4.2.1 Delay at First Output Queue

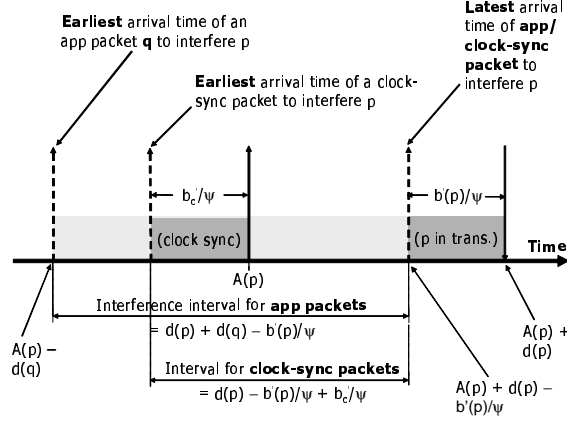
Consider an application packet  $p$  that arrives at the first output queue at a source node  $i$  (denoted  $OutQ_1$  in Figure 8(a)). In [14], we show that the upper bound on the total delay incurred by  $p$  in this queue is given by:

$$O_1^i(p) = \sum_{q \in P_i} \left[ \frac{X(p) + X(q)}{w(q)} \right] a(q) \delta_i + \left[ \frac{X(p)}{\theta} \right] \delta_i \quad (1)$$

where  $P_i$  is the upper bound on the number of packets that can arrive at the output queue for outward transmission and  $\delta_i$  is the aggregate worst-case, execution, dispatching, queue-transfer time of UPA (on source node  $i$ ) for scheduling, dispatching, and transferring the packet from the output queue.

The upper bound  $O_1^i(p)$  on packet delay derived in [14] is *sufficient, but not necessary*. This is because, the upper bound  $O_1^i(p)$  is established in [14] by observing that any packet  $q$  will be scheduled by UPA before packet  $p$ , only if  $q$  arrives no sooner than  $A(p) - X(q)$  and no later than  $A(p) + X(p)$ , where  $A(p)$  denote  $p$ 's arrival time at the output queue.

This interference interval for packet  $p$ ,  $I(p) = [A(p) - X(q), A(p) + X(p)]$ , which has a length  $X(p) + X(q)$ , is only sufficient for a packet  $q$  to interfere with the transmission of packet  $p$ . Packet  $q$  can interfere with packet  $p$  only if  $q$  arrives during this interval. That does not imply that  $q$  will *always* interfere with  $p$ , if  $q$  arrives during this interval. It only implies that for  $q$  to interfere with  $p$ , it must arrive during this interval. But even if  $q$  arrives during this interval, it may very well not interfere with  $p$ . *This is precisely due to UPA's scheduling order.*



**Figure 9. UPA's Packet Interference**

For example, if  $q$  were to arrive after  $A(p) + X(p) - X(q)$  but before  $A(p) + X(p)$ , the absolute termination time of  $q$  will occur after that of  $p$ . Under EDF,  $q$  will then be scheduled after  $p$ , since  $q$  has a longer absolute deadline (or termination time) than that of  $p$ . However, under UPA, it is quite possible that  $q$  can be scheduled before  $p$ , because  $q$  may have an extremely large utility than  $p$  and thereby  $q$  may appear earlier in UPA's schedule than  $p$  (due to UPA's re-rankings).

Figure 9 illustrates the packet interference situation under UPA.

Developing a necessary and sufficient interference interval will require schedule construction. To avoid this and still obtain a “tighter” interference interval, we consider the notion of an “optimistic” interference interval during which a packet  $q$  may interfere with packet  $p$  with very likelihood.

Observe that UPA sorts packets by decreasing order of pseudo-slopes. Thus, we regard that if the pseudo-slope of packet  $q$  is larger than that of packet  $p$ , for all times during  $p$ 's (sufficient) interference interval  $I(p)$ , then  $q$  has a very high likelihood for interfering with  $p$ 's transmission. Thus, if  $\Gamma(p, t)$  denotes the pseudo-slope of packet  $p$  at time  $t$ , then our first optimistic interference condition becomes:

$$\Gamma(q, t) > \Gamma(p, t), \forall t \in I(p) \quad (2)$$

Another key step in UPA's scheduling process is examining adjacent pairs of packets in the schedule, computing  $\Delta$ , and swapping the packets, if the reverse order can lead to higher local aggregate utility. Thus, we regard that if  $\Delta_{q,p}(t) \geq 0$  for all time instants  $t$  during the interval  $I(p)$ , then again packet  $q$  has a very high likelihood for interfering with  $p$ 's transmission. This

becomes our second optimistic interference condition:

$$\Delta_{q,p}(t) \geq 0, \forall t \in I(p) \quad (3)$$

We thus determine an *optimistic* upper bound on packet  $p$ 's delay in the first output queue at a source node  $i$  using Equation 1 by considering all packets  $q \in P_i$ , only if  $q$  satisfies Equations 2 and 3.

#### 4.2.2 Delay at Second Output Queue

The delay incurred by a packet at the second output queue (denoted  $OutQ_0$  in Figure 8(a)) can be similarly derived, except for two issues that will affect the delay. These include: 1. The input arrival rate of packets into the second output queue will now change, as it will depend upon the rate at which packets will be output from the first output queue; and 2. packet transmission times on the outgoing network link (from the source node to the next intermediate node) must be taken into account.

For a packet  $p$  that can arrive at the first output queue of a source node  $i$  for a maximum of  $a(p)$  times during  $w(p)$ , the rate at which the packet will be output (from the first output queue) is given by:

$$R_1^i(p) = \left\lceil \frac{O_1^i(p)}{w(p)} \right\rceil a(p) \quad (4)$$

This will be the rate at which  $p$  will arrive at the second output queue.

For a clock synchronization packet  $c$ , the arrival rate at the second output queue is given by:

$$R_1^i(c) = \left\lceil \frac{O_1^i(c)}{\theta} \right\rceil \quad (5)$$

Thus, the (optimistic) upper bound on the delay incurred by a packet  $p$  to arrive at the next intermediate node, since its arrival at the source node's second output queue is given as:

$$O_0^i(p) = \sum_{q \in P_i} \left[ X(p) + X(q) - \frac{b'(p)}{\varphi} \right] R_1^i(q) \left[ \frac{b'(q)}{\varphi} + \delta_i \right] + \left[ X(p) - \frac{b'(p)}{\varphi} + \frac{b'_c}{\varphi} \right] R_1^i(c) \left[ \frac{b'_c}{\varphi} + \delta_i \right] \quad (6)$$

#### 4.2.3 Total Delay at a Node

Thus, the (optimistic) upper bound on the total delay incurred by a packet  $p$  to arrive at the next intermediate node, since its arrival at a source node  $i$ 's first output queue is given as:

$$N_i(p) = O_1^i(p) + O_0^i(p) \quad (7)$$

Observe that an (optimistic) upper bound on the total delay incurred by a packet  $p$  to arrive at a node  $j$  (which can either be an intermediate node or be the packet’s destination node), since its arrival at *any* intermediate node  $i$ ’s first (input) queue (denoted  $InQ_0$  in Figure 8(b)) is the same as that given by Equation 6. This is because the two queues at an intermediate node (denoted  $InQ_0$  and  $OutQ_0$  in Figure 8(b)) directly correspond to the two queues at a source node (denoted  $OutQ_1$  and  $OutQ_0$  in Figure 8(a)).

### 4.3 LocDUCE Algorithm

The key heuristic employed by the algorithm is to allocate channels for messages in decreasing order of their potential “return of investments.” The return of investment for a message is simply the timeliness utility that can be obtained when the message is delivered to the destination.

To obtain an estimate of the maximum possible return of investment from a message, LocDUCE first allocates channels to each message, assuming that the messages will not interfere with each other (i.e., under zero contention between message packets). Thus, the algorithm considers the network as an un-weighted graph (or equivalently, all edges have the same weight), where each vertex represents a system node and an edge represents a connection between a pair of nodes. For each message, LocDUCE determines the shortest path-distance from the source to destination (i.e., one with the smallest hop-count) by running the breadth-first-search (BFS) algorithm. The path (or channel) with the shortest path-distance for a message will yield the maximum possible utility for the message, since all TUFs considered are *non-increasing*.

Note that this utility is the theoretically maximum possible utility and cannot be achieved in practice because of message contention.

LocDUCE computes channels for messages in decreasing order of the maximum possible message utility. For the  $k^{th}$  message (in the decreasing maximum possible utility order), denoted  $m_k$ , the algorithm first updates the network graph with the channel of the  $k - 1^{th}$  message, denoted  $m_{k-1}$ , computed in the previous step i.e., for each node  $i$  that lie in message  $m_{k-1}$ ’s channel, LocDUCE includes message  $m_{k-1}$  in the set  $P_i$  of messages that can arrive at node  $i$  for outward transmission. Thus, once the set  $P_i$  of each node  $i$  in  $m_{k-1}$ ’s channel has been updated, a message channel that includes any outgoing edge from any node  $i$  in  $m_{k-1}$ ’s channel may suffer interference from  $m_{k-1}$ .

LocDUCE now runs Dijkstra’s shortest-path algorithm to determine the shortest path-distance channel for message  $m_k$ . Again, the shortest path-distance channel will yield as maximum utility for the message as possible, since all TUFs are non-increasing. For determining the shortest

path-distance, the weight of any outgoing edge from any node  $i$  in  $m_{k-1}$ 's channel is determined using Equation 7. Note that Equation 7 gives an upper bound on the total delay incurred by the message to arrive at the next intermediate node by travelling through the edge.

---

**Algorithm 4.1** LocDUCE: High Level Description

---

```

1: for all incoming messages do
2:   if DT  $i$  is not registered then
3:     Determine  $U_i(t_d)$  for the DT  $i$  at  $t_d$ 
4:     Register the DT in  $DTList$ 
5:     Order the  $DTList$  in decreasing order of  $U_i(t_d)$ 
6:     for all DTs in the  $DTList$  do
7:       for all existing paths to the destination do
8:         Determine  $D_j(p)$  for the DT
9:         Choose the link with lowest  $D_j(p)$ 
10:        Select next hop node  $j: D_j(p)$  is minimum
11:        Store the routing decision
12:        Forward message to node  $j$ 
13:     else
14:       Forward Packet using the stored route

```

---

The algorithm repeats the process for each message  $m_i \in M, i \in [1, n]$ .

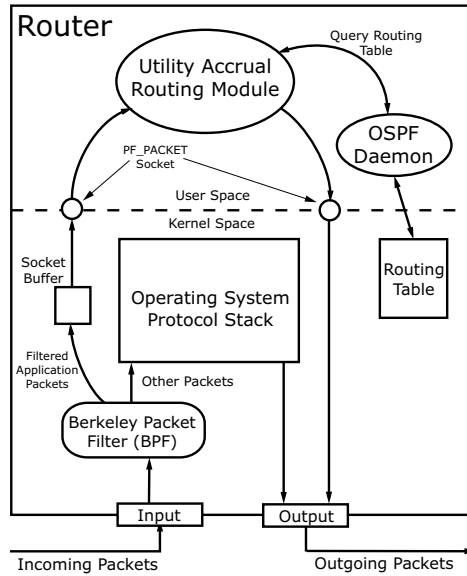
Note that while LocDUCE computes the channel for message  $m_k$ , it considers the potential interference that  $m_k$  can suffer from messages  $m_i, 1 \leq i \leq k - 1$ . However, it does not consider the interference that  $m_k$  can cause on messages  $m_i, 1 \leq i \leq k - 1$  and accordingly update their channels that were computed in earlier steps. This is precisely due to the heuristic nature of the algorithm. LocDUCE ignores such “backward” interference and reasons that it will not be significant, as each message considered for channel establishment at any given time is the one with the largest (theoretically) maximum possible utility from the remaining unexamined ones. Further, correcting such backward interferences will be computationally expensive. Thus, the algorithm takes this heuristic approach and seeks to maximize the total utility attained by all messages as much as possible.

A high level description of LocDUCE is described in Algorithm 4.1.

## 5 Implementation

We implemented a prototype of LocDUCE in the application layer of the Linux operating system. The implementation includes: (1) an application-layer UA routing module (URM), and (2) MAC-layer UA packet scheduler (in the Linux kernel). The routing module captures messages off the interface. To circumvent the actual forwarding by the module of the operating system, a firewall is configured to drop all messages. The captured messages are processed and

proper routes are selected by the URM. Once the routing process is completed, the messages are placed on to the output interface directly using the Libnet library<sup>3</sup>.



**Figure 10. Implementation Architecture of the Utility Accrual Routing Module**

The routing module uses the PF\_PACKET type socket, to capture messages from the interface and to process them. In order to limit the number of messages that are captured, a Berkeley Packet Filter (BPF) or Linux Socket Filter (LSF) [28] is installed on the socket. Toward this, the `setsockopt()` call is used with the `SO_SOCKET_FILTER` option. The filter selects only the application messages from the other messages arriving at the interface. Figure 10 shows how the entire architecture of the implementation is structured.

An open source implementation of the OSPF routing protocol was used to obtain the routing table for the operation of the routing module. The URM queries the OSPF daemon for the routing table and uses this routing table for routing.

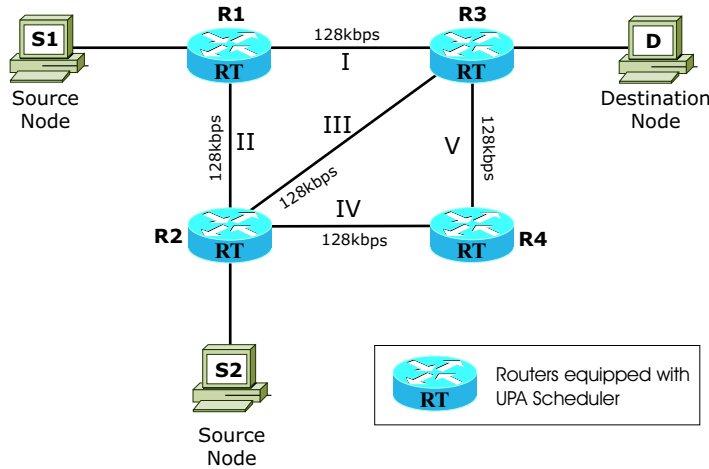
## 5.1 Routing Module

The routing module is multi-threaded. The module starts a separate thread for each interface of the router to handle arriving messages. The module maintains a `MUTEX` protected list of applications and associates each application with an output interface on which the messages are routed. When a message from a distributable thread arrives for the first time, it is registered with the router and the route is determined. From the next message onwards, unless there is a new thread arrival, the stored route is used to forward the message.

<sup>3</sup>Libnet library version 1.1.0 can be obtained from <http://www.packetfactory.net/libnet/>

## 6 Experimental Evaluation

We implemented the UA routing daemon on Redhat Linux machines running the 2.4.18 version of the kernel. In the implementation, the routing module is a “user process” performing the routing functionality. The test-bed comprises of Linux machines configured as routers and interconnecting subnets, to form a Wide Area Network (WAN). The subnets contain source nodes that host segments of distributable threads. The threads invoke operations on remote objects and thus generate real-time traffic. The network also includes a destination node that hosts remote objects and thus receives real-time traffic.



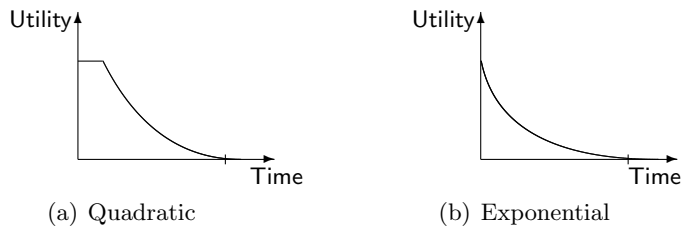
**Figure 11. Network Topology Used in Experimental Study**

Figure 11 shows the topology of the network that was used in the experimental study. The machines  $S_1$  and  $S_2$  are the source nodes and  $D$  is the destination. The machines  $R_1$ ,  $R_2$ ,  $R_3$ , and  $R_4$  are the routers.

We used the Hierarchical Token Bucket (HTB) [29] queuing discipline to limit the bandwidth to 128 kbps for fair comparison of the various schemes. The links connecting the source machines to the routers are not constrained to eliminate the contention among real-time messages on the link from the source node to the edge router. The clocks of all machines are synchronized using the Network Time Protocol (NTP) [27].

### 6.1 Experimental Settings

We consider six TUFs in our study. These include the step, soft-step, linear, and MITRE/TOG AWACS association TUFs shown in Figures 2(a), 2(b), 2(c), and 3, respectively, and the quadratic and exponential TUFs shown in Figures 12(a) and 12(b), respectively.



**Figure 12. Quadratic and Exponential TUFs**

Note that the soft-step TUF has a shape similar to that of the GD/CMU plot correlation and track maintenance TUFs. Our motivation to consider the linear, quadratic, and exponential TUFs is that they are close variants of the AWACS TUF. In fact, in the design of the AWACS TUF, the designers empirically derived the slope of the TUF [24]. Therefore, we consider TUFs that are similar to the AWACS TUF, but with different slopes. We also consider the step TUF in our study as that represents the traditional deadline constraint.

**Table 1. Experimental Parameters**

DTs	TUF Type	Maximum Utility	Termination Time (sec.)	Source Node
DT1	Step	100	4.0	$S_1$
DT2	Exp	100	4.0	$S_2$
DT3	Soft-step	100	4.0	$S_1$
DT4	Linear	100	4.0	$S_2$
DT5	Quad	100	4.0	$S_2$
DT6	AWACS Assocn.	100	4.0	$S_2$

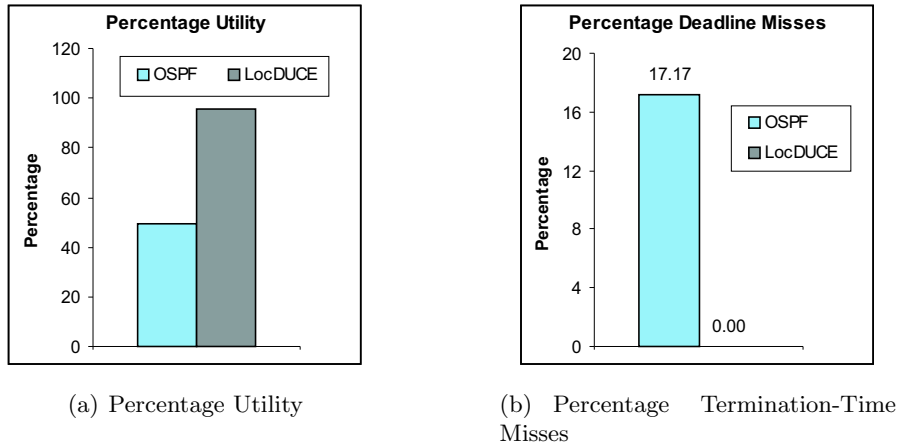
Our experimental scenarios can be classified into: (1) static and (2) dynamic. In the static scenario, message attributes such as data rate, message size, and inter-arrival time remain the same. In the dynamic scenario, we vary attributes including message laxity, arrival rate, message size, and utility variance.

Table 1 shows characteristics of the distributable threads (abbreviated as DTs in the table) considered in our experimental study.

## 6.2 Static Scenario

Figure 13 shows the percentage utility and percentage termination-time misses of LocDUCE and OSPF. Percentage utility is simply the percentage ratio of accrued aggregate utility to the maximum possible utility, and percentage termination-time misses is the percentage ratio of the number of messages meeting their termination times to the total number of messages.

The better performance of LocDUCE with respect to OSPF can be explained as follows: OSPF always selects the shortest path for any message to its destination. For example, for threads 2, 4, 5 and 6, OSPF takes the shortest path, R2-R3 to D. This decision of the algorithm

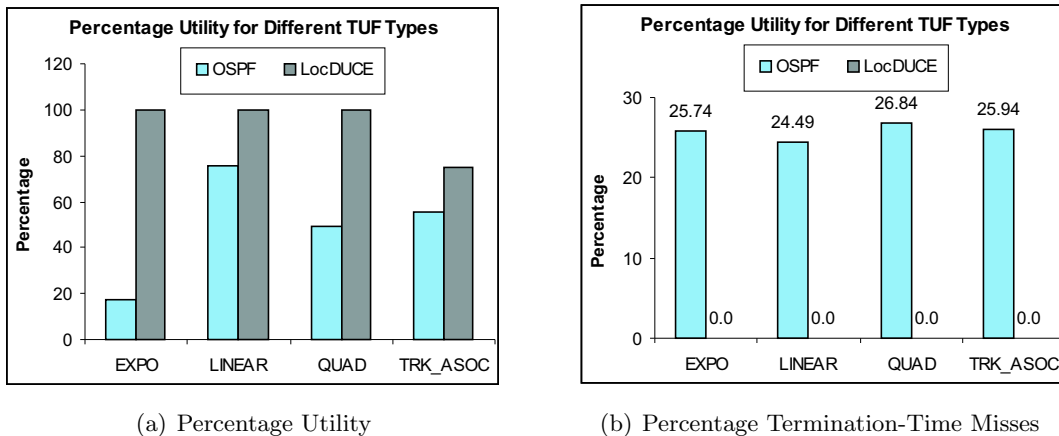


**Figure 13. Performance of LocDUCE and OSPF for Static Scenario**

does not optimize the message flows along all the available paths. Although some versions of OSPF perform load-balancing, the loads along the paths having the same cost are balanced. In cases when there are multiple paths with unequal costs, the algorithm does not divide the flows. Hence, when the shortest path becomes overloaded, the performance degrades for all message flows, irrespective of the TUFs. (Note that First-In-First-Out queues are used in experiments involving OSPF.)

In the case of LocDUCE, the messages traverse diverse paths, one through R2–R3 and another through R2–R4–R3. Thus, the performance is better.

Figure 14 shows the percentage utility and percentage termination-time misses of LocDUCE and OSPF under different TUF types.



**Figure 14. Performance of LocDUCE and OSPF for Static Scenario for Different TUF Types**

Figure 14 shows that LocDUCE performs better than OSPF. We also observe that the utility achieved under some TUFs are significantly lower, thus reducing the overall utility. This is be-

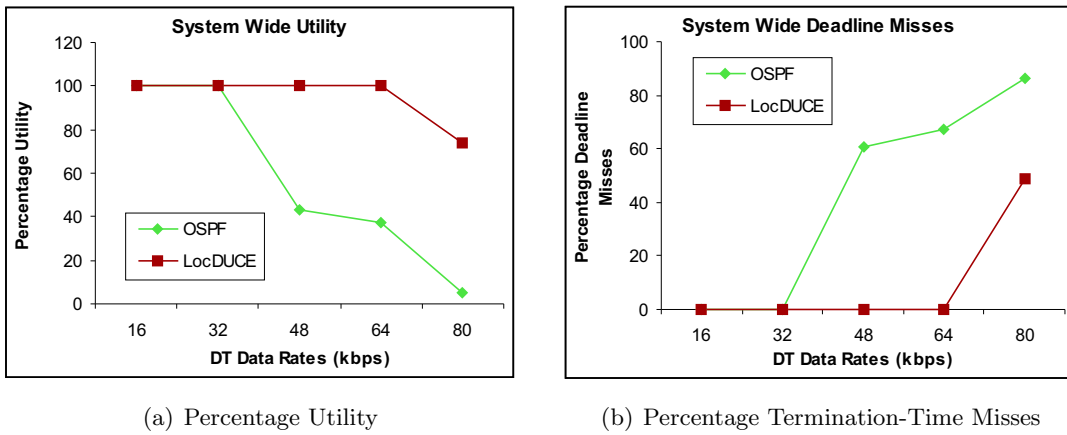
cause there is no differentiation made among the messages that belong to different distributable threads.

### 6.3 Dynamic Scenario

#### 6.3.1 Increasing Packet Arrival Rate

This class of dynamic scenario experiments were conducted with four distributable threads that originate at node S2 and invoke remote operations on node D. We increased the message arrival rate from 2 packets/sec to 10 packets/sec in steps of two.

Figure 15 shows the percentage utility and percentage termination-time misses of LocDUCE and OSPF under increasing message arrival rates. Table 2 shows the average and standard deviation values for the experiments.



**Figure 15. System Wide Percentage Utility and Deadline Miss Comparison**

From Figure 15, we observe that LocDUCE performs significantly better than OSPF. Note that the data rate indicated is the individual rate for the messages. Thus, the actual load on the link is four times that value. Therefore, even under such heavily loaded conditions, we observe that LocDUCE performs better.

**Table 2. Average and Standard Deviation**

DT Data Rates (kbps)	Percentage Utility				Percentage Termination-Time Misses			
	OSPF		LocDUCE		OSPF		LocDUCE	
	<i>Avg.</i>	<i>Dev.</i>	<i>Avg.</i>	<i>Dev.</i>	<i>Avg.</i>	<i>Dev.</i>	<i>Avg.</i>	<i>Dev.</i>
16	99.96	0.0024	99.97	0.002	0.00	0.00	0.00	0.00
32	99.94	0.0314	99.95	0.001	0.00	0.00	0.00	0.00
48	43.33	0.0600	99.94	0.002	60.787	0.366	0.00	0.023
64	37.51	0.0978	99.93	0.001	86.15	0.450	0.00	0.015
80	5.050	0.1248	73.84	0.495	90.08	0.547	48.65	0.056

The good performance of LocDUCE over OSPF is due to the fact that there are multiple paths to the destination, which are not explored by OSPF. We also observe that all message flows suffer the same performance degradation under OSPF. This is due to the fact that OSPF does not prefer one message flow over another and the scheduling is strictly first-in-first-out.

From Table 2, we observe that the standard deviation of percentage utility and termination-time miss values are low. This indicates the consistency of the algorithm performance.

#### 6.4 Overhead Analysis

Overhead for LocDUCE includes: (1) channel establishment overhead and (2) runtime overhead. The channel establishment overhead is due to the fact that every new distributable thread has to be registered. Once the threads are registered, routes may have to be re-established for all other registered threads.

Let  $k$  be the number of routes stored for each destination. Let  $n$  be the maximum number of threads registered on each interface of the router making the decision. Since each router has to determine the lowest delay interface among  $k$  different routes and among each of the  $n$  threads registered on the interface, the complexity of determining the route is  $O(nk)$ .

The runtime overhead is the overhead of fetching the stored route for a given thread.

#### 6.5 Performance Under Heavily Loaded Downstream Links

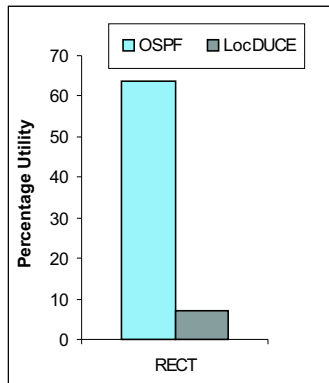


Figure 16. Performance Under High-Load Downstream Links

There are some conditions under which OSPF may perform better than LocDUCE even though the performance difference may be limited to particular message flows. LocDUCE performs route allocation based on the “local” information regarding the message flows (i.e., flows that are maintained in the list). When the downstream links to the router are heavily loaded

(as it routes messages generated from some other source), the performance of the messages may degrade.

Figure 16 shows one such case. The figure shows a distributable thread with a step TUF having a better performance under OSPF than LocDUCE. This is precisely because of the previously discussed condition.

## 7 Conclusions and Future Work

Real-time CORBA 2.0 distributable threads that use multi-hop networks as their underlying platform and are subject to utility accrual timeliness optimality criteria require coherent, utility accrual, node- and network-level resource management. This includes employing utility accrual scheduling algorithms for managing node-level resources and utility accrual channel establishment algorithms for managing network channel resources.

In this paper, we present a utility accrual channel establishment algorithm called LocDUCE that establishes real-time channels for inter-node messages of distributable threads. LocDUCE allows messages to inherit TUF time constraints from their parent threads and establishes channels, maximizing the messages' total attained utility. Our experimental measurements using a prototype implementation reveal that LocDUCE accrues significantly higher utility than OSPF.

There are several directions for further research. One direction is to decentralize LocDUCE and thus develop a distributed UA channel establishment algorithm for improved performance. Another direction is to develop UA channel establishment algorithms that can tolerate node and link failures.

## Acknowledgements

This work was supported by the US Office of Naval Research under Grant N00014-00-1-0549.

## References

- [1] OMG, "Real-time corba 2.0: Dynamic scheduling specification," Object Management Group, Tech. Rep., September 2001, oMG Final Adopted Specification, <http://www.omg.org/docs/ptc/01-08-34.pdf>.
- [2] E. D. Jensen and J. D. Northcutt, "Alpha: A Non-Proprietary Operating System for Large, Complex, Distributed Real-time Systems," in *Proc. of The IEEE Workshop on Experimental Distributed Systems*, 1990, pp. 35–41.
- [3] The Open Group Research Institute's Real-Time Group, *MK7.3a Release Notes*. Cambridge, Massachusetts: The Open Group Research Institute, October 1998.

- [4] D. Wells, "A Trusted, Scalable, Real-Time Operating System," in *Proc. of The Dual-Use Technologies and Applications Conf.*, 1994, pp. 262–270.
- [5] GlobalSecurity.org, "Multi-platform radar technology insertion program," <http://www.globalsecurity.org/intell/systems/mp-rtip.htm/>.
- [6] —, "Bmc3i battle management, command, control, communications and intelligence," <http://www.globalsecurity.org/space/systems/bmc3i.htm/>.
- [7] E. D. Jensen, "Asynchronous Decentralized Real-time Computer Systems," in *Real-Time Computing*, W. A. Halang and A. D. Stoyenko, Eds. Springer Verlag, October 1992.
- [8] P. Li, B. Ravindran, *et al.*, "A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints," <http://nile.ece.vt.edu/submissions/GUS-TOC03.zip>, August 2003, submitted to IEEE Trans. on Computers (under review).
- [9] C. D. Locke, "Best-Effort Decision Making for Real-time Scheduling," Ph.D. dissertation, Carnegie Mellon University, 1986, CMU-CS-86-134.
- [10] R. Clark, "Scheduling Dependent Real-time Activities," Ph.D. dissertation, Carnegie Mellon University, 1990, CMU-CS-90-155.
- [11] G. Koren and D. Shasha, "D-Over: An Optimal On-line Scheduling Algorithm for Overloaded Real-Time Systems," in *Proc. of IEEE Real-Time Systems Symp.*, December 1992, pp. 290–299.
- [12] K. Chen and P. Muhlethaler, "A Scheduling Algorithm for Tasks Described by Time Value Function," *Journal of Real-Time Systems*, vol. 10, no. 3, pp. 293–312, May 1996.
- [13] D. Mosse, M. E. Pollack, and Y. Ronen, "Value-Density Algorithm to Handle Transient Overloads in Scheduling," in *Proc. of IEEE Euromicro Conf. on Real-Time Systems*, June 1999, pp. 278–286.
- [14] J. Wang and B. Ravindran, "Time-Utility Function-Driven Switched Ethernet: Packet Scheduling Algorithm, Implementation, and Feasibility Analysis," *IEEE Trans. on Parallel and Distributed Systems*, vol. 15, no. 2, pp. 119–133, February 2004.
- [15] D. Ferrari and D. C. Verma, "A Scheme for Real-time Channel Establishment in Wide Area Networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, pp. 368–379, April 1990.
- [16] A. Banerjea, D. Ferrari, B. A. Mah, M. Moran, D. C. Verma, and H. Zhang, "The Tenet Real-time Protocol Suite: Design, Implementation, and Experiences," *IEEE/ACM Trans. on Networking*, vol. 4, no. 1, pp. 1–10, February 1996.
- [17] D. D. Kandlur and K. G. Shin, "Design of a Communication Subsystem for HARTS," University of Michigan, Tech. Rep. CSE-TR-109-91, December 1991.
- [18] K. G. Shin and C. C. Chou, "Design and Evaluation of Real-time Communication for Field Bus Based Manufacturing Systems," *Proc. of the IEEE Local Computer Network Symp.*, pp. 483–492, September 1992.
- [19] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time Communication in Multi-hop Networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.
- [20] K. G. Shin, C. C. Chou, and S. K. Kweon, "Distributed Route Selection for Establishing Real-time Channels," *IEEE Trans. on Parallel and Distributed Systems*, vol. 11, no. 3, pp. 318–335, March 2000.
- [21] C.-J. Hou, "Routing Virtual Circuits with Temporal QoS Requirements in Virtual Path-based ATM Networks," *IEEE Trans. on Computers*, vol. 48, no. 11, pp. 1228–1243, 1999.
- [22] Q. Zheng and K. G. Shin, "Fault-tolerant Real-time Communication in Distributed Computing Systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 5, pp. 470–480, May 1998.

- [23] G. Manimaran, H. S. Rahul, and C. S. R. Murthy, "A New Distributed Route Selection Approach for Channel Establishment in Real-time Networks," *IEEE Trans. on Networking*, vol. 7, no. 5, pp. 698–709, October 1999.
- [24] R. Clark, E. D. Jensen, *et al.*, "An Adaptive, Distributed Airborne Tracking System," in *Proc. of IEEE Workshop on Parallel and Distributed Real-Time Systems*, ser. LNCS, vol. 1586. Springer-Verlag, April 1999, pp. 353–362.
- [25] D. P. Maynard, S. E. Shipman, *et al.*, "An example real-time command, control, and battle management application for alpha," Carnegie Mellon University, Archons Project TR-88121, December 1988.
- [26] G. L. Lann, "Proof-Based System Engineering and Embedded Systems," in *Lecture Notes in Computer Science*, G. Rozenberg and F. Vaandrager, Eds. Springer-Verlag, October 1998, vol. 1494, pp. 208–248.
- [27] D. L. Mills, "Improved Algorithms for Synchronizing Computer Network Clocks," *IEEE/ACM Trans. on Networks*, vol. 3, pp. 245–254, June 1995.
- [28] G. Insolubile, "The Linux Socket Filter: Sniffing Bytes Over the Network," *Linux Journal*, Issue 86, June 2001, <http://oceanpark.com/webmuseum/linuxjournal/issue86-june2001/>.
- [29] M. Devera, "Hierarchical Token Bucket (HTB) Packet Scheduler," <http://luxik.cdi.cz/~devik/qos/htb/>.