A Real-Time Performance Comparison of Distributable Threads and Event Channels

Yuanfang Zhang, Bryan Thrall, Stephen Torri, Christopher Gill, and Chenyang Lu
Department of Computer Science and Engineering
Washington University, St. Louis, MO, USA
{yfzhang,thrall,storri,cdgill,lu}@cse.wustl.edu

Abstract

No one middleware communication model completely solves the problem of ensuring schedulability in every DRE system. Furthermore, there have been few studies to date of the trade-offs between alternative middleware communication models under different application scenarios. This paper makes three contributions to the state of the art in middleware for distributed real-time and embedded systems. First, it describes what we believe is the first example of integrating release guards directly with CORBA distributable threads to ensure appropriate release times for sub-tasks along an end-to-end computation. Second, it presents empirical results in which release guards improve schedulability of distributable threads compared to a greedy protocol in which arriving tasks simply begin to run as soon as they can. Third, we offer the first empirical comparisons of the distributable thread and event channel models under three different communication scenarios and then using a randomized workload.

1. Introduction

In standardization efforts such as the OMG's Real-Time Common Object Request Broker Architecture (RTCORBA) specification 1.2 [11][1] , the Real-Time Specification for Java (RTSJ) [2], and the Distributed Real-Time Specification for Java [9], two main models have emerged: thread-based and event-based. In the thread-based model, threads traverse object methods making invocations on other (possibly distributed) object methods. In the event-based model, distinct events are passed to event handlers that are executed upon receipt of the event.

For end-to-end scheduling of CORBA systems, neither the distributable thread communication model nor the event-channel communication model completely solves the problem of ensuring schedulability of all tasks in an entire distributed system. In particular, distributable threads do not follow the regimented structure of method invocations seen with the event-channel model, but rather follow computationally determined paths that can vary significantly at run-time. For highly dynamic systems, distributable threads can be a more appropriate abstraction due to the expected cost of tearing down and setting up event subscriptions dynamically. However, in more static systems, distributable threads add the risk that any jitter in their execution latency may have significant repercussions for periodically scheduled systems.

Meeting the deadlines of tasks throughout a *distributed* system is a relatively well studied research topic in the field of real-time systems, and yet there remain several important open questions, which this paper addresses:

1. Can the fundamentally dynamic nature of thread-based middleware communication models be made more predictable through integration of inter-processor synchronization mechanisms such as release guards [15]?
2. What are the performance costs of release guards on distributable threads and event services?
3. What are the relative performance implications of thread-based and event-based communication models for three canonical communication topologies: (1) sequential chains of tasks, (2) branching graphs of tasks and (3) dynamic task graphs?
4. What are the relative performance implications of thread-based versus event-based communication models.

In order to answer these questions, the paper provides the first empirical performance comparisons between Event Channels [6] and Distributable Threads [16] in RT-CORBA middleware.

Experimental platforms: All the experiments described in this paper were performed using ACE/TAO version 5.1.4/1.1.4 on a testbed consisting of four ma-

1 The RTCORBA 2.0 specification was recently renumbered as RTCORBA 1.2, but its contents remained the same.

chines. Two of them are Pentium-IV 2.5GHz machines, and the other two are Pentium-IV 2.8GHz machines. Each of them has 500MB RAM and 512KB cache, and runs version 2.4.22 of the KURT-Linux operating system [3]. These platforms provide a CPU-supported timestamp counter with nanosecond resolution.

The data stream user interface (DSUI) and data stream kernel interface (DSKI) are provided with the KURT-Linux distribution. The DSUI is used to record information from the middleware and application layers, while the DSKI is used to collect information at the kernel level like context switching. By using both DSUI and DSKI instrumentation, we can obtain a precise accounting of task start and stop times, thread context switches, CPU idle intervals, and other relevant events across multiple system levels. By then integrating the information from both the DSKI and the DSUI, we can give a reasonably exact execution time for any subtask or thread. Figure 1 is a histogram of the aggregate subtask execution time for the first subtask of the EC model in the Static Graph Test (see Section 4.2). The execution times largely fall in the narrow window around 50 milliseconds, the target execution time for the subtask.

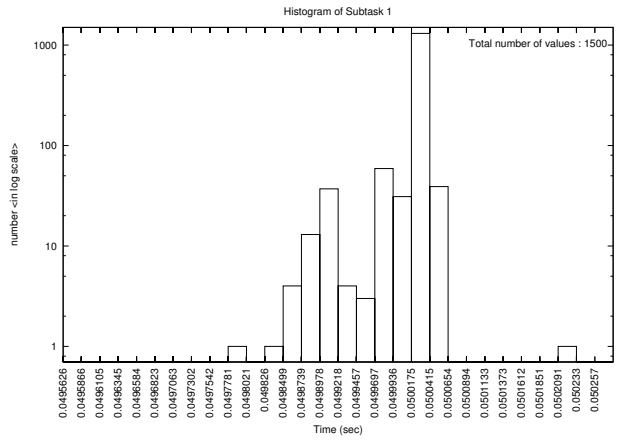

Figure 1: Actual execution times for subtask

Paper overview: This paper is structured as follows. Section 2 first gives an overview of the previously developed implementations of the distributable thread and event channel models this paper considers. Section 3 then describes our implementation of release guard mechanisms for distributable threads and federated event channels in TAO, and presents a simple example showing the benefits of those mechanisms. Section 4 presents a set of overhead comparisons between Event Channels and Distributable Threads for three canonical communication topologies. Section 5 presents the design and empirical results of one experiment designed to simulate a more complex task set, using a randomly generated workload. Section 6 surveys other research related to the work presented in this paper. Finally, Section 7 offers conclusions, including observations and recommendations based on our results.

2. Distributable Thread and Event Channel Implementations

We based the experiments described in this paper on the distributable thread and event channel implementations provided in TAO. In both cases we used the Kokyu dispatching framework to provide real-time dispatching of distributable threads and events. In both the distributable thread and event channel experiments presented in this paper, Kokyu is configured to use the nonpreemptive EDF scheduling algorithm [8].

2.1. Distributable Threads

In Real-Time CORBA 1.0 systems, application end-to-end timeliness requirements must be acquired from the client, propagated with the invocation, and delivered to the server. Dynamic real-time systems may require more information than just priority, such as deadline, time of execution and laxity [11]. Hence, the communication model is required to characterize the schedulable entity and subsequently associate parameters with it so that it can be scheduled appropriately. A natural communication model suggested by CORBA's control flow programming model is a thread that can execute operations in objects without regard for physical node boundaries. In Real-Time CORBA 1.2, this communication model is termed a Distributable Thread (DT) [16]. Each DT has a unique system wide ID. It may have one or more execution scheduling parameters, e.g., priority, deadlines and importance.

As a first step, we consider the Static Chain Test described in Section 4.1. In Figure 2, one distributable thread DT_1 executes on three processors P_1, P_2, and P_3. In P_1, a local thread on behalf of the distributable thread DT_1 periodically executes the first subtask, then invokes the second subtask on the processor P_2 by sending a one-way call. In all our tests, we assume the release time of each first subtask is always strictly periodic. Another local thread on P_2 that is waiting for DT_1 begins to execute the second subtask after receiving the one-way call. When this local thread finishes, it again propagates DT_1 to processor P_3 to invoke the third subtask. Each local thread that executes on one local processor on behalf of DT_1 inherits all properties of the DT such as the DT identifier and timeline requirements. It is scheduled and dispatched by the local EDF scheduler and Kokyu dispatcher based on its inherited information.

Although TAO has both preemptive and nonpreemptive dispatching mechanisms, we only enable nonpreemptive dispatching in Section 4 and 5 for a fair comparison with

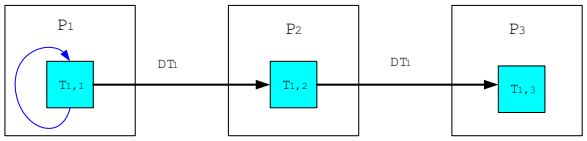

Figure 2: DT chain test structure

Event Channels. We also use nonpreemptive dispatching for the example in Section 3.4 which shows the effect of RGs.

To propagate end-to-end timeline requirements with each request, we need a separate QoS descriptor for each task, which is used to store at a minimum an unique ID, deadline, and execution time. The QoS descriptor is sent along with every invocation from the client to the server.

2.2. Event Channel

The CORBA [12] Event Service [6] implements the Mediator design pattern [4]. The nodes of the distributed communication network are classified as *suppliers* and *consumers* according to their roles in the Event Channel. Suppliers register the types of events they produce (supply) with the Event Channel. Consumers subscribe to the types of events on which they rely. The Event Channel acts as an intermediary between suppliers and consumers so that they do not need to know explicitly about each other.

The concept of an Event Service has been extended by RT-CORBA to account for the quality-of-service (QoS) requirements of real-time systems [12]. The QoS parameters of suppliers and consumers may be specified to the Event Channel so that it can distribute events according to those requirements.

The TAO Event Channel [6] applies the RT features described in [12] to the CORBA Event Service [10]. The TAO EC adds guarantees for real-time events dispatching and scheduling, through the Kokyu Dispatching Framework [5]; centralized event filtering and correlation; and periodic processing support to the CORBA Event Service [6].

Our Event Channel tests use a Federated Event Channel [6] to coordinate the processors in which subtasks execute. Each processor has its own EC, and the ECs exchange events via a Gateway, as described in [6], which acts as a Consumer of the supplier's events and a Supplier of the consumer's events. The Gateway can reside in either processor.

The motivation for using a Federated Event Channel is to dispatch locally without needing to communicate with remote nodes, which is also the DT's behavior. To satisfy this requirement, all events need to be dispatched locally to the processor where the job will be executed. EC dispatching happens inside each individual EC, so each processor must have its own EC.

Each subtask in our EC tests is implemented as a supplier-consumer pair. The supplier pushes events which trigger the subtask execution in a single consumer. If the subtask is the first subtask in a task, then the supplier has an associated timeout handler registered with a reactor to receive timeout events. When the timeout handler receives a timeout event, it triggers the supplier to push an event to the consumer. Then when the consumer finishes executing the subtask, it notifies the supplier for the next subtask, which behaves as if it had been triggered by a timeout handler. Of course, the final subtask's consumer has no supplier for the next subtask.

For example, consider the Static Chain Test described in Section 4.1. The structure of timers, suppliers, and consumers on processors P_1, P_2, and P_3 is shown in Figure 3. In P_1, a Supplier1_1 with a timer (depicted as a clock) pushes events (arrows) to a Consumer1_1, which then executes subtask $T_{1,1}$ and pushes another event to the processor P_2 through another Supplier1_2. P_2's Consumer executes subtask $T_{1,2}$ then pushes yet a third event to P_3, whose Consumer executes the final subtask, $T_{1,3}$. This sequence of events happens periodically, every time the timer determines that it is time for T_1 to execute again.

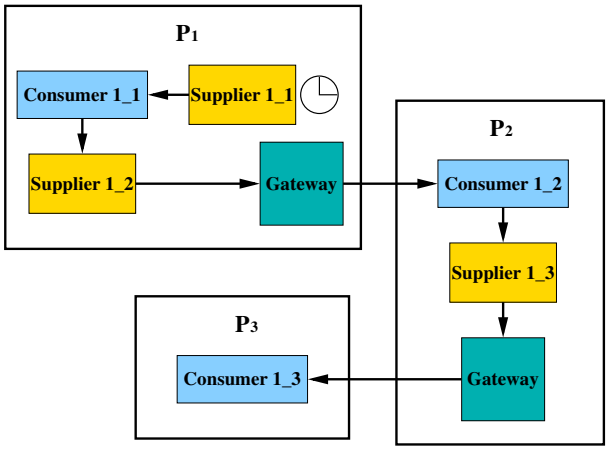

Figure 3: EC chain test structure

The gateway that connects two Event Channels which process consecutive subtasks, due to the limitations of the gateway implementation, is treated as a single supplier to the consumer's EC and specifies a fixed period as part of its subscription on that EC. However the actual periodicity of the events pushed through the gateway depends entirely on the suppliers on the other side. This means that the deadline and periodicity used by EDF to determine the utility of an event pushed from a Gateway might be skewed from what it should be given the actual run-time QoS of the subtask the event is triggering. The gateway can be instantiated in either processor, but for our implementation it is on the supplier side.

Comparing Figure 2 with Figure 3, for the same Chain Test, developers who choose the EC model need to manipulate all the consumers and suppliers, while the DT model users only need to take care of one distributable thread. The DT model is more convenient to understand and implement in the system with end-to-end scheduling tasks.

3. Release Guard Implementation

The Release Guard protocol [15] ensures that the time between two consecutive releases of the same subtask is not less than the period. This ensures that we can analyze system schedulability assuming each subtask is periodic. This behavior adds little overhead since the protocol only needs to know the period of a task and the last time the subtask was released. As Sun mentioned in his doctoral thesis [14], the following three rules are used to update the release guard g for subtask T.

1. g is initially equal to 0.
2. When an instance of subtask T is released, update g to the current time plus the period p of T.
3. Update g to the current time if the current time is a processor idle point on the processor where T executes.

These rules force the distributed application to maintain the following property: the release guard protocol releases a job of T at a time equal to g, or when the immediate predecessor of T completes, whichever is later.

3.1. Release Guard in Distributable Threads

We implemented release guards for DTs in TAO as a slight modification of the Kokyu Dispatching Framework [5]. For one request sent from Client to Server, it only needs to be guarded on the Server side. Specifically, we use a hash map in Kokyu's Dispatcher to record the latest release time for each subtask. Whenever the scheduler in the server processor receives a request from the client, it first checks this map before inserting the request into the dispatching queue. If the current time is earlier than the request's latest release time plus the period, the request blocks itself on a condition variable and sets a timer which will fire at the proper time. Otherwise, it is inserted into the dispatching queue immediately. When the timer of a blocked request fires, the blocked request is inserted into the dispatching queue. The current time is also recorded in the hash map as a new latest actual release time for the next job. Because the hash map may be accessed from multiple threads, a mutex is used to protect its integrity. The hash map and condition variables are used to reduce jitters in end-to-end response times by implementing the first two rules of the release guard protocol described in Sun's thesis [14], and we show their effects in Figure 5. The complexity of the hash map is $O(1)$.

Moreover, in the preemptive implementation of Kokyu Dispatching, a scheduler thread does the dispatching. Whenever a subtask finishes its job, the scheduler thread will wake up and pick up the current highest priority subtask in the ready queue to run. When it finds the ready queue is empty, but some subtasks are waiting at the release guard, it will release one subtask immediately and allow it to make use of the CPU "idle" time. With the help of a specific scheduler thread, our DT release guard implements Sun's third rule, and improves the end-to-end response time. However, since this third rule is not required for proper functioning of the release guard (it is only an optimization for improving average response time), we do not enable it in all our tests.

3.2. Release Guard in Event Channel

Note that the release guard (RG) is not bound with the Event Channel. Although the timeout event and all consumer trigger events are passed through the EC, they are not subjected to the RG algorithm. Only the events triggered by the gateway are subjected to the RG when passing through the EC. The notification of a consumer to the supplier of the next subtask does not go through the EC, so it does not go through the RG either.

Release Guards are also implemented in the Event Channel as a slight modification of the Kokyu Dispatching Framework [5]. Before an event is enqueued in one of Kokyu's dispatching lanes, the Release Guard checks that the event is being released at an appropriately periodic time. To avoid priority inversion, there is a Release Guard for each of the dispatching lanes.

Let r_i be the latest release time of event i (stored in a mutex-protected map); then the target release time of r_{i+1} is $r_i + p$, where p is the period of the event. If the Release Guard determines that the event is being released at an appropriate time, then it allows the event to be enqueued; otherwise, the Release Guard sets a timer to go off at $r_i + p$ and buffers the event in the Release Guard Queue. When the timer goes off, a Release Guard Thread removes the event from its queue and enqueues it in the Kokyu dispatching lane. A Dispatch Thread always removes the event at the head of the lane and dispatches it.

Note that for the EC we have not implemented the Release Guard behavior for the third rule (just as this behavior is not enabled for DTs in this paper); testing with this behavior implemented is left for future work. Because we are using Federated Event Channels, this means that an event travelling from one processor to another will be forced through the Event Channel twice: once when it is pushed to the gateway through the local EC, and once when the gate-

way pushes it to the remote Consumer through the remote EC. It would be more appropriate to only perform the Release Guard on the remote EC, because a Release Guard only needs to be performed once to reduce jitter, and there is less time for jitter to accumulate again before the event reaches its destination consumer if Release Guard is performed within the EC of the destination processor. We modifies the Kokyu framework to distinguish between these two cases, so we can restrict the RG to performing only when dispatching gateway pushed events.

3.3. System Model

In our experiments, we only consider end-to-end periodic tasks, which are sequences of subtasks. Each subtask of the task consists of component jobs which execute in sequence on different processors to form a tree. Each component job requires only resources local to the processor on which the job executes.

The system contains m processors, P_j for $j = 1, 2, ..., m$, and n periodic tasks, T_i for $i = 1, 2, ..., n$. Each task T_i has $n(i)$ subtasks, $T_{i,k}$, for $k = 1, 2, ..., n(i)$. We say that $T_{i,k}$ is the immediate predecessor of $T_{i,k+1}$. Each subtask $T_{i,k}$ is represented by a tuple $(e_{i,k}, p_{i,k}, a_{i,k})$, where $e_{i,k}$ is the worst case execution time needed to execute subtask $T_{i,k}$, $p_{i,k}$ is the period of the parent task T_i, and $a_{i,k}$ is the phase of the subtask $T_{i,k}$ (the default value of $a_{i,k}$ is 0). The phase of Task T_i means the release time of the first job of its first subtask ($a_{i,1}$). The phase of any following subtask is zero. The relative end-to-end deadlines of the tasks are equal to their respective periods.

3.4. Effect of Release Guards

The system in this example contains two processors and three periodic tasks implemented using distributable threads. Tasks $T_1 = (2, 4, 0)$ and $T_3 = (1.8, 3, 5.1)$ each only has a single subtask, and they execute on P_1 and P_2, respectively. Task T_2 has two subtasks: $T_{2,1} = (2, 6, 0)$ executes on P_1, and $T_{2,2} = (2, 6, 0)$ executes on P_2. The subdeadline $D_{i,k}$ of each subtask $T_{i,k}$ is equal to the end-to-end absolute deadline D_i of its parent task T_i. In this example, the nonpreemptive EDF scheduling policy is used, and only the first two rules of the RG protocols in Sun's thesis are enabled. Figure 4 shows the schedule of the tasks when they are synchronized according to the greedy protocol in the DT's implementation. In contrast, the schedule in Figure 5 is seen when the release guard protocol is used.

As shown in Figure 5, the bi-directional arrow denotes where the release guard delayed the release of a subtask. Job $J_{2,2,2}$ was delayed by the release guard. Otherwise Job $J_{3,1,2}$ which had a higher priority but released later than Job

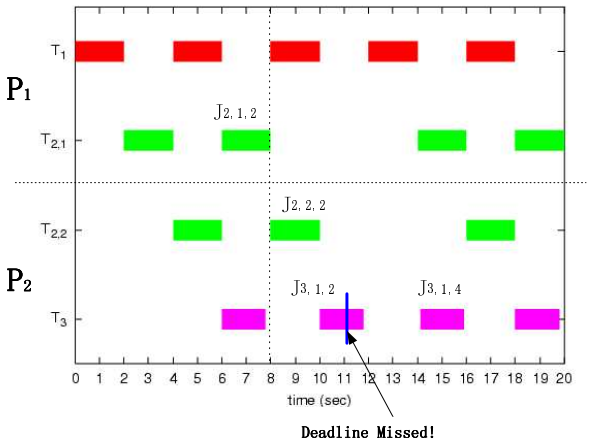

Figure 4: Scheduling with greedy protocol

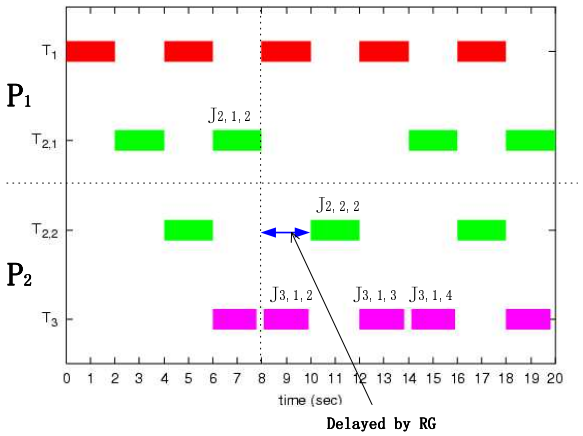

Figure 5: Scheduling with release guard

$J_{2,2,2}$ can not begin to execute until 10th second as shown in Figure 4. Finally Job $J_{3,1,2}$ missed its deadline, and the next job of Task 3 was also cancelled because of the strict periodic restriction. Furthermore, the end-to-end response time of Task 2 was always 6 seconds in Figure 5 while it can be 4 or 6 seconds when using the greedy synchronization protocol in Figure 4. These experimental results show the profile of how the release guard works in the middleware. The release guard really can improve the predictability and schedulability of end-to-end tasks.

4. Benchmark Results on Overhead

To empirically compare two communication models under different scenarios, we need to calculate for both of them how much time is wasted on the additional activities except the job execution. We call the total time wasted on all additional activities the overhead. To isolate and analyze the overhead more easily, all three tests in this section model one periodic task T_1 in the system. The task T_1 has 2 or 3 subtasks, each on a separate processor. The pe-

riod for the task is 200 milliseconds and the execution time for each subtask is 50 milliseconds. The deadline of each subtask is equal to the period of its parent task T_1. Each test is active for 300 seconds, so the task will be run 1500 times. All the time values shown in tables are measured per job run with nano-second resolution. We run each test five times and show the average, maximal and minimal values of the overhead for both EC and DT. In addition, the non-preemptive EDF scheduling policy is used in all three tests in this section and the random workload chain test in Section 5. All the machines we use in the tests have 500MB RAM and 512KB cache, and also run version 2.4.22 of the KURT-Linux operating system.

4.1. Static Test Chain

In this test, the task T_1 is a linear chain of subtasks, which has three subtasks $\{T_{1,1}, T_{1,2}, T_{1,3}\}$, each on a separate processor $\{P_1, P_2, P_3\}$. We call them Client, Middle and Server, respectively. Client is a Pentium-IV 2.5GHz machine. Middle and Server are both Pentium-IV 2.8GHz machines.

For the EC implementation, the Middle and the Server processors ran for slightly longer (305 and 310 seconds, respectively) to accommodate startup delays (the Server processor is started first, then the Middle, and finally the Client). However, since the Client initiates task execution, the active period remains 300 seconds. We can see the extra 5 and 10 seconds for the other two processors as slightly longer times. For the DT implementation, the Client processor sends a shutdown request to the Middle and the Server processors after it finishes the 300 seconds of work. The Middle and the Server processors shut themselves down when they receive the request from the Client, which should be earlier than EC.

Since we let the DSUI record the program start and stop events from the application layer, the total running time is decided. The DSUI also records the start and stop time for any job at the application layer, and the DSKI collects information like context switching at the kernel level. By integrating the information from both the DSKI and the DSUI, we can give a reasonably exact execution time for any subtask by filtering any context switching between the DSUI pairs. The number shown in the first column is the average execution time for one job. Moreover, since the DSKI has a special thread to record the CPU idle time, the idle time in the second column is decided only by the running time of that thread per job run. The total minus the execution time and the idle time leaves the number in the last column. We call it Overhead & Misc, which includes the overhead from both the middleware layer and the operating system layer for per job run and some trivial unaccounted time. Misc also includes the time for recording DSUI events. To decrease the influence from DSUI, we always keep recording same number of DSUI events for both EC and DT in all our tests. In most cases, the EC's and DT's DSUI overheads are comparable with each other. However, the system thread could introduce variation in other systems.

Table 1: EC/DT in static chain test

EC/DT	Execution Time (ms) Avg.	Idle Time (ms) Avg.	Overhead & Misc (ms)		
			Avg.	Max.	Min.
Client	50.0474/ 50.0268	149.6016/ 149.6183	0.3512/ 0.3495	0.3561/ 0.3537	0.3489/ 0.3465
Middle	50.0171/ 50.0317	152.7737/ 150.8956	0.5437/ 0.6443	0.5466/ 0.6497	0.5388/ 0.6399
Server	50.0159/ 50.0277	156.2965/ 152.3614	0.3638/ 0.3530	0.3708/ 0.3586	0.3543/ 0.3494

Analysis of Overhead on the Middle Side: We see the overhead difference on the Client and the Server processors are really trivial, but the overhead difference on the Middle processor is notable. We try to explain it by doing analysis about the overhead on the Middle processor for both two model. The EC and DT have different implementation mechanisms. It is hard to compare them side by side, but from the following analysis of the overhead, we can get useful information about which parts cost more in each of them. EC has three working threads on the middle side: one is a main thread which waits for periodic events from the Client side and dispatches them, second is a release guard thread which inserts the delayed events into the dispatching queue; third is a dispatching thread whose main job is to execute the second subtask and push the third subtask to the Server periodically. During each run of 300 seconds, both EC and DT tests ran the second subtask 1500 times. In each iteration, the main thread did two jobs: waited for events from the Client; pushed the events to local EC. Meanwhile, the EC dispatching thread did the following three jobs: First, the consumer received a event pushed by the main thread. Second the consumer executed the second subtask and pushed an event to the associated supplier, and then the supplier pushed the event to the gateway through the EC. Third the gateway pushed the event to the Server processor. The overhead for each thread is shown in Table 2. Moreover, to record the important time points which interest us, we rerun the same test with more DSUI events. The total overhead increased slightly in Table 2, compared with the corresponding value in Table 1.

The overhead for the dispatching thread in the DT example can be divided into five main parts. First the dispatching thread waits for the requests from the Client. Second is the EDF Scheduling time. Third is the Kokyu dispatching time. Fourth is the time for forwarding requests the proper ser-

Table 2: Client overhead analysis for EC

Total Overhead		0.5554 (ms)
Main Thread	Waiting for Requests	0.0870 (ms)
	Push Events to EC	0.0638 (ms)
Dispatching Thread	Dequeuing Requests	0.0293 (ms)
	Push Events to EC	0.0516 (ms)
	Oneway Call Sending	0.0815 (ms)
Release Guard Overhead		0.0925 (ms)
Unaccounted		0.1497 (ms)

vants. Fifth is for sending the oneway call to the next processor after executing the local subtask.

Table 3: Client overhead analysis for DT

Total Overhead		0.6654 (ms)
Dispatching Thread	Wait for Requests	0.2100 (ms)
	Scheduling Time	0.0725 (ms)
	Dispatching Time	0.0191 (ms)
	Forward Requests	0.0200 (ms)
	Oneway Call Sending	0.2432 (ms)
Release Guard Overhead		0.0201 (ms)
Unaccounted		0.0805 (ms)

Comparing the above two tables, we find DT pays more time than EC for sending a oneway call from one processor to another, because DT needs to propagate more information with the oneway call, such as an unique ID, deadline, execution time, etc., which costs extra overhead. On the other hand, DT pays less for Release Guard mechanism. There are two reasons that explain this: First, for each event delay, EC needs to do two extra context switches between the dispatching thread and the release guard thread; Second, the enqueuing and dequeuing from the release guard queue also cause overhead. Moreover, each scheduling plus dispatching in the EC costs less time than that in the DT, but one request needs to pass through EC more than once on each processor and every time it will be scheduled and dispatched. For any request, the DT only schedules and dispatches it once on each processor. Overall, DT's additional overhead for propagating more timing information dominates the total overhead, making it less efficient than EC.

4.2. Static Task Graphs

This test models task T_1 which has multiple subtasks. Some of them may be performed in parallel. In this case, the first subtask $T_{1,1}$ triggers the execution of two successors $T_{1,2}$, $T_{1,3}$ at almost the same time. The subtasks $T_{1,2}$ and $T_{1,3}$ only rely on subtask $T_{1,1}$, and do not have interdependence. Essentially, this test models a simple branching task such as might be used in a fault-tolerant distributed system. $T_{1,1}$ executes in the Client processor and $T_{1,2}$ and $T_{1,3}$, which may be copies of the same subtask or two totally different subtasks, execute in two Server processors. The client processor is a Pentium-IV 2.5GHz machine and the two Server processors are Pentium-IV 2.8GHz machines.

For the EC and DT implementation, the Server processors ran for slightly longer (305 seconds) to accommodate startup delays (Two Server processors are started first, then the Client). However, since the Client initiates task execution, the active period remains 300 seconds. We can see the extra 5 seconds for the two Server processors as slightly longer idle times. Since two Server processors of the DT finish earlier than the EC's, the total running time for the DT example is shorter than that of EC.

Table 4: EC/DT in static graph test

EC/DT	Execution Time (ms) Avg.	Idle Time (ms) Avg.	Overhead & Misc (ms)		
			Avg.	Max.	Min.
Client	50.0412/ 50.0279	149.4942/ 149.4043	0.4653/ 0.5627	0.4690/ 0.5770	0.4622/ 0.5487
Server1	50.0065/ 50.0278	152.9738/ 152.8189	0.3625/ 0.3519	0.3723/ 0.3552	0.3497/ 0.3491
Server2	50.0093/ 50.0267	152.9683/ 151.7602	0.3564/ 0.3475	0.3604/ 0.3548	0.3522/ 0.3424

Results: In this test we observed a greater overhead difference on the Client processor. The DT implementation must make serial function calls to multiple servers. Although the Event Channel also sends events sequentially to two servers, the EC model costs less overhead than the DT model for each send. Based on this result and the previous analysis of overhead on the Middle side, we conclude if we further increase the number of requests, the overhead difference on the Client processor should increase. It looks like that the EC model is more suitable for the static branching scenario.

When we compare the overhead of the static task graph with that of the static task chain on the Client, we find the static task graph incurs more overhead. That is because on the Client side the static task graph needs to send two requests to two independent machines instead of one. But for Server machines, either in static task graph or static task chain, their overheads look similar.

4.3. Dynamic Task Graphs

This test models a task T_1, which has multiple subtasks, one of which, due to its importance to the system, needs assurance that it can be completed independent of the situation on a particular processor. So, when the system is in Fault Tolerant (FT) mode, the subtask, $T_{1,2}$, is sent to multiple processors (the Primary and Secondary Middle proces-

sors) and executed on each. In this case, $T_{1,2}$ is duplicated and one copy is sent to each Middle processor ($T_{1,2a}$ goes to the Primary Middle processor and $T_{1,2b}$ goes to the Secondary Middle processor). The two Middle processors provide a degree of fault tolerance on the system in case either of them is unable to complete the subtask within its QoS constraints. When the system is in normal, non-FT mode, T_1 behaves the same as the task in the Static Task Chain test. That is, $T_{1,2}$ executes only on the primary Middle processor (Middle1). The application switches between the two modes randomly with 50% probability in each mode. This test examines the dynamic behavior of a distributed real-time system by measuring the overhead of the system when it is switching.

In our experiment, we have four machines, as shown in Figure 6. Subtasks $T_{1,1}$ and $T_{1,3}$ execute on processors P_1 and P_4, respectively. When the system operates in FT mode during the test, subtask $T_{1,2}$ is duplicated and executed on both processors P_2 and P_3. For the normal mode, subtask $T_{1,2}$ is only executed on processor P_2. Processor P_4 begins to execute subtask $T_{1,3}$ after receiving a synchronization signal from either P_2 or P_3, and skip the other.

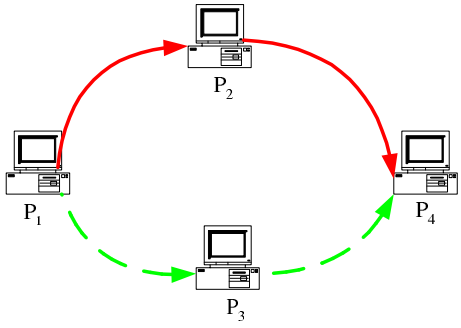

Figure 6: Dynamically changing topology test

For the EC and DT implementations, the Server and two Middle processors ran for slightly longer (310 and 305 seconds, respectively) to accommodate startup delays (the Server processor is started first, then both Middle processors, and finally the Client processor). However, since the Client initiates task execution, the active period remains 300 seconds. We again see the extra 5 and 10 seconds for the other two processors as slightly longer idle times. Also the DT tests finished earlier than the EC's.

Results: The Middle2 processor only receives about 727 jobs from the Client, which is around the half of the total 1500 jobs. All the time values shown in Table 5 are per job of the task. Since Middle2 receives fewer jobs, the processor is more likely to remain idle and wait for next requests. This actual result is consistent with our expectation. Since the Client in this test sent out more requests than that in the static chain test but fewer than that in the static graph test, the overhead on client side was also between those two. In this dynamic graph test, the EC needed to subscribe to two different event types at the beginning statically for two modes. When the mode switches, it changes the type of the event it pushes. This "static" subscription to two types of events requires additional programming effort and must be performed at programming time, while the DT can really dynamically switch by just sending one more request or not. Our experience indicates that the DT is more suitable for highly dynamic applications that need to reconfigure task graphs online.

Table 5: EC/DT in dynamic graph test

EC/DT	Execution Time (ms) Avg.	Idle Time (ms) Avg.	Overhead & Misc (ms)		
			Avg.	Max.	Min.
Client	50.0137/ 50.0276	149.5311/ 149.5092	0.4457/ 0.4578	0.4556/ 0.4619	0.4362/ 0.4537
Middle1	50.0205/ 50.0354	153.3422/ 152.6661	0.5946/ 0.6786	0.5986/ 0.6885	0.5919/ 0.6747
Middle2	50.0399/ 50.0347	369.1480/ 365.8382	0.5788/ 0.7190	0.5825/ 0.7229	0.5769/ 0.7157
Server	49.9207/ 50.1496	156.7210/ 154.0328	0.4810/ 0.5149	0.4867/ 0.5208	0.4715/ 0.5100

5. Random Workload Performance Results

5.1. Random Workload Generation

Through three preliminary testing results in Section 4, we established an initial characterization of the DT and EC implementations. Now we need to verify it in more complex and real applications. In our next experiment, each system had 4 processors and 12 tasks. We first generated the period for each task, which is uniformly distributed between 50msec and 1sec. The number of subtasks in each task is randomly distributed between 1 and 4. Subtasks are uniformly assigned to processors as long as any two subtasks in a same task are not assigned to the same processor. The CPU utilization is uniformly distributed between 0.5 and 0.8. On each processor, all the subtasks randomly divide the CPU utilization. Specially, to determine the utilization of individual subtasks on a processor, we generated a random number between 0.001 and 1 for each subtask and called it the utilization factor of the subtask. We then divided the utilization factor of each subtask by the sum of the utilization factors of all subtasks on the same processor, to obtain a normalized utilization factor. The utilization of a subtask is the product of the processor utilization and its normalized utilization factor. The execution time of each subtask is simply its utilization multiplied by its period which is same for all subtasks in one task.

5.2. Random Workload Test

In addition to the sources of overhead difference mentioned in Sections 4, the EC and DT Random Workload tests add several more. The DT implementation must spawn a separate thread for each subtask which runs in the processor, while the EC implementation does not require additional threads for increasing the number of subtasks.

The Random Workload Test attempts to mimic the behavior of a more complex, realistic system. The generated task set and its QoS is detailed in Table 6. We denote the four end systems "Node1", "Node2", "Node3", and "Node4". Figure 7 shows how the twelve tasks of this test are distributed among the end systems. According to our assignment rule, we do not allow two subtasks of a particular task to execute on the same end system.

Table 6: Random task chain parameters

Subtask	Execution Time (msec)				Period (msec)	Phase Delay (msec)
$T_{1,\{1,2,3,4\}}$	94	32	122	125	849	335
$T_{2,1}$	12				627	10
$T_{3,1}$	3				199	80
$T_{4,\{1,2,3,4\}}$	30	30	17	23	257	132
$T_{5,1}$	34				952	501
$T_{6,1}$	3				896	312
$T_{7,\{1,2,3,4\}}$	13	5	12	9	110	26
$T_{8,\{1,2\}}$	160	46			752	481
$T_{9,1}$	143				838	277
$T_{10,\{1,2,3\}}$	64	59	107		703	673
$T_{11,1}$	121				916	442
$T_{12,\{1,2,3\}}$	15	7	11		190	168

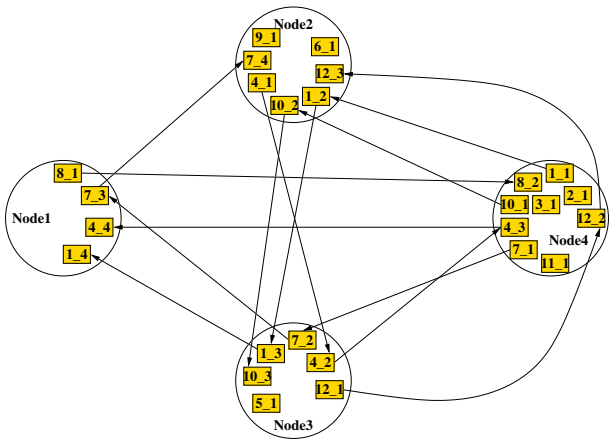

Figure 7: Random workload subtask distribution

All the tasks in this test are periodic, and the test is active for 300 seconds. Results from the EC and DT implementations are shown in Table 7. Since the random workload test is composed of 12 tasks, it is difficult to analysis the overhead for each task. The time values in Table 7 are the totals for all 12 tasks in the system in 300 seconds.

For the DT implementation, each Node runs for about 340 seconds, with a small part of the initialization time used for establishing the connections between nodes. a large interval is needed for synchronizing the start time of tasks on four Nodes. At the start time, all the Nodes spawn task threads and are ready to execute subtasks. Our measurement only begins after the synchronization phase and continues until four nodes all run about 300 seconds.

For the EC implementation, each Node runs for approximately 340 seconds, 40 of which are used for negotiating connections between Nodes and synchronizing the start time for actual task execution. All the Nodes create their gateways to each of the other Nodes, as described in Section 2.2, then arrange a start time. At the start time, all the Nodes begin the timeouts which trigger event pushes that cause subtasks to be executed.

Because of the small phase delays between tasks, it is necessary to arrange for all the processors to start allowing tasks execution at as close to the same time as possible; in this way, the pattern of which subtasks are executed when on a particular end system is much closer to the ideal pattern derived from the theoretical QoS constraints given for the test. Also our measurement begins at that starting time.

Table 7: EC/DT in random workload chain test

EC/DT	Execution Time (sec)	Idle Time (sec)	Overhead & Misc (sec)
Node1	167.3970/ 166.8722	134.8827/ 130.3003	2.1251/ 2.8340
Node2	165.4350/ 164.8024	135.0996/ 132.5027	2.1384/ 2.7016
Node3	170.9234/ 170.8312	127.0672/ 124.7414	2.7394/ 4.4198
Node4	193.6443/ 194.3415	102.5973/ 102.0207	2.7233/ 3.6588

Results: We find that for the random chain test, the overheads difference between the EC and the DT is nontrivial. Although the DT and EC showed the close performances in previous tests, DT's multithreaded mechanism causes the difference in this test. For each subtask on a machine, DT needs to generate and maintain a working thread, and these threads compete at every scheduling point. Contrarily, the EC maintains only three threads for any test.

6. Related Work

Synchronization algorithms: Several synchronization algorithms are related to the use of release guards in this work. The *greedy synchronization* protocol is concerned with releasing and executing all sub-tasks as soon it is possible to do so [8]. This protocol is typically a common feature in many of today's non-real time operating system schedulers. Despite the protocol's ability to ensure minimal release delay, it can lead to bursty task behavior. Bettati introduced a set of protocols based on *phase-modification* [1]. The phase-modification protocol can be coupled with General Fixed-Priority scheduling[1] to produce a schedule of tasks. The *release-guard* protocol [15] ensures that the time between two subtasks is not less that the period. This ensures that the releasing of the subtasks remains as periodic possible. This behavior adds little overhead since the protocol only needs to know the period between tasks and the last time the subtask was released. An alternative approach to synchronization in DRE systems is distributed priority ceiling [13]. That approach has not been adopted in standard-based DRE middleware. It is not addressed in this paper.

Distributable Threads in Operating Systems: Distributable threads were originally introduced in the Alpha Kernel [7]. While the Alpha Kernel implemented distributable threads at the operating system level, our work is on middleware: distributable thread and event channel implementations of end-to-end activities.

Event Channel and Distributable Thread in DRE Middleware: Both communication models have been implemented on DRE middleware and evaluated *separately* [6] and [16]. To our best knowledge, our work is the first empirical comparison between the two models. Moreover, none of the earlier EC or DT implementations support Release Guards which are important for end-to-end scheduling.

7. Conclusions

In this paper we have described the integration of release guards with distributable threads and federated event channels, to support predictable scheduling of end-to-end tasks in DRE middleware. We have systematically compared the performance of distributable threads and federated event channels for both random workloads and three canonical communication topologies: sequential task chains, branching task graphs and dynamic task graphs. In all of our experiments, both communication models introduced small overhead per subtask indicating that both models are reasonably efficient for many DRE applications. From the results, EC is slightly more efficient than DT in each of our benchmark tests using a *single* end-to-end task, especially for the branching graph scenario. Furthermore, DT performs worse when the system ran more concurrent tasks. This is because DT requires more concurrent threads which introduce higher context switch overhead. Our empirical results indicate that, from a performance perspective, EC is more efficient for applications with static end-to-end tasks, while DT provides a more suitable programming model for dynamic changing the task topology at run-time.

References

[1] Riccardo Bettati. *End-to-end scheduling to meet deadlines in distributed systems*. PhD thesis, Department of COmputer Science, University of Illinois at Urbana-Champaign, 1994.

[2] Greg Bollella, James Gosling, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[3] Douglas Niehaus, *et al.*. Kansas University Real-Time (KURT) Linux. `www.ittc.ukans.edu/kurt/`, 2004.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (The Gang of Four. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] Chris Gill, Douglas C. Schmidt, and Ron Cytron. Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 91(1), January 2003.

[6] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of OOPSLA '97, Atlanta, GA*, pages 184–200, 1997.

[7] J. D. Northcutt. The Alpha Operating System: Kernel internals. Archons Project Technical Report, 1988.

[8] Jane W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[9] E. Douglas Jensen. Rationale for the Direction of the Distributed Real-Time Specification for Java: Panel Position Paper. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Newport Beach, CA, March 2002. IEEE/IFIP.

[10] Object Management Group. *CORBA Event Service Specification*, December 1997.

[11] Object Management Group. Real-time CORBA Specification 2.0. Object Management Group, February 2003.

[12] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 3.0.1 edition, February 2003.

[13] L. Sha, R. Rajkumar, and J. Lehoczky. Real-time synchronization protocol for multiprocessors. In *RTSS*, 1988.

[14] Jun Sun. *Fixed-Priority End-to-End Scheduling in Distributed Real-Time Systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.

[15] Jun Sun and Jane W.-S. Liu. Synchronization protocols in distributed real-time systems. In *International Conference on Distributed Computing Systems*, pages 38–45, 1996.

[16] Yamuna Krishnmaurthy. Real-time CORBA 2.0: Dynamic Scheduling. OOMWorks LLC, 2002.