

Choir: A Real-Time Middleware Architecture Supporting Benefit-Based Proactive Resource Allocation

Peng Li, Binoy Ravindran, Jingtang Wang, and Glenn Konowicz
Real-Time Systems Laboratory
Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061, USA
{peli2, binoy, jiwang5, gkonowicz}@vt.edu

Abstract

Asynchronous real-time distributed systems are inherently non-deterministic. To deal with such non-determinism's, we have developed a family of proactive resource management algorithms that support benefit-function based, end-to-end QoS management. This paper describes a middleware implementation of these algorithms, called Choir. The Choir middleware allows the user express the task end-to-end timeliness requirements using Jensen's benefit functions. Furthermore, the middleware system can transparently replicate, and possibly migrate the computational subtasks to conquer uncertainties such as workload fluctuations, changes of system resources, so that the system aggregate benefit is maximized. Initial experimental results suggest the effectiveness the Choir middleware.

1 Introduction

Asynchronous real-time distributed systems are emerging in many domains including defense, telecommunication, and industrial automation for the purpose of strategic mission management [8]. Such systems are fundamentally distinguished by the significant run-time uncertainties that are inherent in their application environment and system resource states [8]. Consequently, it is difficult to postulate upper bounds on application workloads for such systems that will always be respected at run-time. Thus, they violate the deterministic foundations of hard real-time theory, which ensures that all timing constraints are always satisfied under deterministic postulations of application workloads and execution environment characteristics.

To deal with such non-determinism's, we have developed a family of *proactive* resource allocation

algorithms, such as those presented in [6, 9, 5]. These algorithms are proactive in the sense that they allow *user-triggered* resource allocation for *user-specified*, arbitrary, application workload and host failure patterns. Our simulation results have shown that these algorithms perform well in terms of maximizing system aggregate benefit. In the meanwhile, we are interested in prototyping these algorithms and determining how they would help to design adaptable and dependable asynchronous real-time distributed systems.

The recent advance of middleware technologies, that enables communication and coordination in a distributed computing system provides the ideal vehicle to implement QoS-aware resource management algorithms. Examples of the real-time middleware include the TAO [13] implementation of the OMG Real-Time CORBA specification, and other non CORBA-compliant real-time middleware, such as the ARMADA project [1] and our own DeSiDeRaTa real-time middleware [12]. However, none of the existing middleware architecture supports the concept of *proactive* resource allocation, which is the key idea embedded in our algorithms. Furthermore, since we are considering end-to-end timing constraints specified using Jensen's benefit functions, the middleware system must allow the user to express the benefit-based timing constraints, transfer these constraints along the path of execution in the system, and always allocate resources in respect of these constraints. The recently approved Real-Time CORBA 2.0 Dynamic Scheduling specification [4] defines a framework that precisely addresses these requirements. For example, it defines the interfaces "for assigning, discovering, and altering the dynamic scheduling parameters." However, implementations of the new RT-CORBA specification are not available yet. Another difficulty with the existing middleware architecture is due to their inadequate scheduling service sup-

port. Our proactive resource management algorithms use best-effort scheduling algorithms, such as the DASA scheduling algorithm [3], for both process and network packet scheduling. These algorithms are not available, or can be easily implemented in these existing systems.

In this paper, we present the *Choir* real-time middleware architecture that supports proactive resource allocation for benefit-based asynchronous real-time distributed systems. The *Choir* middleware implementation serves as a proof of concept; it shows how proactive resource allocation may be employed to maximize the aggregate benefit in asynchronous real-time distributed systems. Our experimental results demonstrate the effectiveness of the *Choir* middleware.

The rest of the paper is organized as follows: Section 2 presents a brief review of the background, e.g. models and algorithms used in the *Choir* middleware. Followed by the descriptions of the individual middleware components in Sections 4, 5, 6, 7, and 8, Section 3 presents an overview of the middleware system. We show the experimental results in Section 9. Finally, the paper concludes with a summary of the work, its contributions, and future work in Section 10.

2 Background: Models and Algorithms

The target application of the *Choir* real-time middleware is a set of end-to-end tasks, either periodic or aperiodic. In general, each periodic task is triggered by events external to the system, such as the arrival of a sensor report message; each aperiodic task is triggered by a periodic task, called its “triggering” periodic task. Furthermore, each task is assumed to consist of a set of subtasks (executable programs) that execute “serially.” The subtasks of a task may execute on different host machines, depending upon the resource allocation decision. In addition, we assume that the task timeliness requirements are expressed using Jensen’s benefit functions [7]. Note that the task timeliness is specified in terms of end-to-end timing constraints, from the beginning of its first subtask to the end of its last subtask.

As in [6], we assume that the user-anticipated workload is expressed using *adaptation function*. The adaptation function describes the user-anticipated workload for a future time interval, starting from a certain reference time point. In many of the real-time distributed systems [12], the workload can be specified in terms of the number of sensor reports and aperiodic events (or “data objects”) processed and transmitted by subtasks and messages, respectively. This is because the data objects constitute the most significant part

of the application workload.

We assume that application subtasks may be dynamically and transparently *replicated* for sharing workload as well as for tolerating end-host failures. Once a subtask is replicated, the replicas of the subtask may share workload and hence reduce the end-to-end response time.

We consider a switched Local Area Network segment, where each end-host is connected to a unique port of an Ethernet switch through a dedicated full-duplex link. That is, the network is using a star topology. Algorithms and techniques for dealing with other network topologies are being developed.

Given the application, adaptation, and system models, the goal of the algorithms is to maximize the system-wide aggregate benefit. This optimization problem can be shown to be \mathcal{NP} -hard. Our algorithm solution to this problem consists of two sets of heuristic algorithms: resource allocation algorithms that determine the number of replicas for each subtask and their host machine assignment; and resource access protocols such as process and packet scheduling algorithms.

We have developed the RBA* and OBA algorithms [6] to determine the number of replicas for each subtask and their host machine assignment. The RBA* and OBA are heuristic algorithms that compute suboptimal resource allocations in polynomial time. Furthermore, RBA* and OBA are proactive algorithms that allocate resources for a future time interval. However, reaction can be implemented as an instance of the proactive allocation. For example, whenever a host failure is detected, the resource management algorithm reacts to the host failure by allocating the replicas to the alive end-hosts for another future time interval.

As for the resource access protocols, we have developed a MAC layer packet scheduling algorithm, called BPA [14]. BPA executes at the MAC-layer of end-hosts and the switch for selecting packets for outbound transmission. The algorithm considers a packet model, where packets have *non-increasing, unimodal benefit function constraints* and seeks to *maximize* the aggregate benefit that is accrued when packets arrive at their destinations.

3 Overview of Choir

The *Choir* middleware system allows the user to specify task structures and their timing requirements. In general, the user can initiate a resource allocation procedure by providing or updating one or more of the existing models such as the task structure, timing requirements, available system resources, and so on. Once the resource allocation is finished, the application tasks are transpar-

ently replicated through the middleware services, according to the new allocation decision. Recall that the resource allocation is done *proactively* in the sense that the resource allocation algorithm always allocates resources for a future time window. Furthermore, the resource allocation can be triggered at arbitrary time whenever the application or the system model changes. Each trigger will result in a new allocation result and invalidate the previous one.

The current version of the *Choir* middleware system contains a Resource Manager (RM) (including a host failure detector), a Run Time Support Library (RTSL), and a Meta Scheduler (MS) that schedules application tasks. For convenience, an Integrated Development and Monitoring Environment (IDME) has also been developed to construct the system and to visualize performance. Furthermore, the underlying packet scheduling algorithms have been implemented in the network device driver and the switch, which are utilized by the middleware system. The current system is implemented for QNX Neutrino RTOS 6.2 and Redhat Linux 7.3. Specifically, the RTSL and the Meta Scheduler run on top of QNX 6.2 and the RM and the IDME run on top of Redhat Linux. However, that does not mean the middleware services are tied to any particular OS functionality of QNX or Linux. Instead, the RunTime Support Library and the Meta Scheduler can be implemented on top of any POSIX-compliant OS. Our reason for choosing the QNX system is primarily due to its high performance and complete support of the POSIX specifications, including a number of POSIX real-time extensions. As for Linux, it is simply because there is a convenient graphic library, e.g. the TrollTech Qt toolkit, available to develop the IDME.

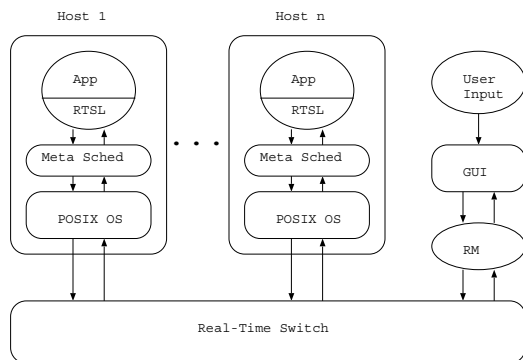


Figure 1. The middleware system.

Figure 1 shows the components of the system and the interaction among them. As discussed in Section 2, the end hosts are connected through a

switch. Since the switch contains real-time packet scheduling algorithms, we refer to it as “Real-Time Switch.” Each end-host runs a POSIX-compliant OS (the current implementation uses QNX 6.2). On top of the OS, the Meta Scheduler (“Meta Sched” in the figure) provides application task scheduling service according to user specified scheduling algorithm. The application is linked with the RTSL and interacts with OS through the Meta Scheduler. Besides the application tasks and the meta scheduler, a host Failure Detection Agent (FDA) also runs on the QNX host machines. The FDA component continually sends heartbeat messages to the resource manager, such that the resource manager can properly react to system resource changes such as host failures or the inclusion of new host machines. A centralized Resource Manager and the IDME run on another machine that is also connected through the Real-Time Switch. The user is responsible for providing task models and their timing requirements into the system. We view the inputs to the application tasks, e.g. data messages, as external to the system, such as from external sensors. Thus, they are not shown in Figure 1. We discuss more details of the components in the sections that follow.

4 The Event-Driven Computational Model

Traditionally, a real-time system consists of a set of time-driven periodic tasks. Additional aperiodic tasks are only allowed to use the spare processor capacity, and hence the schedulability of the periodic task set is preserved. This time-driven model fits well into simple, single-processor, device-level real-time systems, such as direct monitor and control of physical devices. However, in large and complex real-time distributed systems, most of the task executions are triggered by external events, such as a data message arrival from the network. Moreover, due to the distributed nature of the system, even if the initial data are sampled periodically, the release jitter in the later stages of processing becomes so large that the periodic task model does not apply there. Thus, the tasks at a particular processor may arrive at arbitrary time, and may not have known lower bound on the task arrival intervals.

Therefore, an event-driven model is more suitable for real-time distributed systems that are large scale and complex in nature. Here, an “event” refers to an occurrence that causes an execution of a task. Examples of a event include the arrival of a data message to be processed by a task, timer expiry, etc.

In our event-driven model, each application program has a *main* thread, which handles ex-

ternal events and may create child threads in response of these triggering events. Normally the *main* thread creates a child thread for every triggering event. For example, the *main* thread may be blocked on a `receive()` function call, waiting for the arrival of a new data message. Once a data message arrives, the *main* thread creates a new child thread to process the data. Our reason for choosing multi-threading as the basic concurrency mechanism is due to its low overhead. However, threads within different application processes can be scheduled in a system-wide manner. We discuss how the meta-scheduler supports this across-process scheduling in Section 5. Furthermore, since the most significant part of the computation task is executed by the child threads, they become the entities to be scheduled by the meta-scheduler.

5 The Meta Scheduler

The meta-scheduler is a middleware component that supports arbitrary real-time scheduling algorithms on top of any POSIX-compliant OS. The meta scheduler is novel in that it sits on top of any POSIX-compliant OS and interacts with the OS through the standard POSIX APIs. In the meanwhile, the scheduling algorithm can be user-specified and arbitrary. This architecture eliminates the need of modifying the operating system kernel, which may be impossible for commercial operating systems. Details of the meta scheduler are reported in [10]. For completeness, we highlight its key idea and architecture here.

5.1 The Approach

The scheduling problem in essence is a synchronization problem. All child threads need to synchronize their executions with the meta-scheduler, such that at any given time there is only one child thread from an application process is executing. Furthermore, that child thread must be the one selected by the scheduling algorithm. Since a POSIX-compliant OS always schedules threads based on their priorities, it is possible to synchronize threads executions through priority operations.

The key idea of our meta-scheduler architecture is run-time priority mapping and adjusting according to the user-specified scheduling algorithm. The child threads from all application processes are provided with two distinct priorities, e.g. P_1 and P_2 ($P_2 > P_1$). The meta-scheduler dynamically adjusts the priorities of all application child threads in response to a scheduling event. The thread selected by the scheduling algorithm is assigned the higher priority P_2 , while

all other application child threads are assigned the lower priority P_1 . Therefore, once the scheduling decision is made, the selected thread will preempt the execution of all other child threads and possesses the processor. The priority adjustment can be implemented using the POSIX system call `pthread_setschedparam()`.

5.2 Architecture and Code Skeleton

Besides the *main* thread, each application process also contains another thread, called *sched_stub*, to facilitate the dynamic priority adjustment on the application child threads. The *sched_stub* thread is responsible for accepting the meta-scheduler commands and dynamically adjusting the priorities of the threads. The *main* thread of an application process creates the *sched_stub* thread by calling `init_sched()`. Figure 2 shows the code skeleton of an example application *main* thread. After the *sched_stub* is created, the *main* thread enters an infinite loop, waiting for the arrival of new data messages. Once a data message arrives, the *main* thread creates a new child thread to process the data.

```
main()
{
    ... ..
    /* initialize the sched_stub thread */
    init_sched();

    while(1)
    {
        /* blocked here */
        receive(...);
        ... ..
        pthread_create(...,foo,...);
    }
    ... ..
}
```

Figure 2. The main thread skeleton.

Figure 3 shows the skeleton of an example child thread. At the beginning of its execution, the child thread issues the `request_cpu()` primitive (an API supported by the *Choir* middleware), which traps its execution to the underlying meta-scheduler. The `request_cpu()` primitive may be blocked indefinitely until the calling thread is selected to execute by the meta-scheduler. Similarly, at the end of an application thread execution, it issues the `release_cpu()` primitive, notifying the meta-scheduler that the currently running thread has finished execution.

In the current middleware, the *main* thread and `init_sched()` are implemented inside the RTSL. Consequently, the *sched_stub* thread is created by the RTSL.

```

void *foo(void *arg)
{
    /* initialization code */
    ... ..

    /* blocked until this thread is selected */
    request_cpu(...);

    /* process the data */
    ... ..

    /* notify the meta-scheduler it is done */
    release_cpu(...);
}

```

Figure 3. The child thread skeleton.

6 Runtime Support Library

Our *Choir* middleware provides the RunTime Support Library that facilitates communication, replication, and performance monitoring of the application tasks. Furthermore, the RTSL embodies the basic mechanism of the event-driven computation model.

Each application contains the user-provided data processing functions, which are linked with the RTSL. We show a sample user code in Figure 4. The user code should start with a `rt_init_app()` RTSL function call. The `rt_init_app()` function allows the user to register data processing functions to the middleware, so that data messages can be handled by proper functions. To register a data processing function, the user would need to specify a per-host-machine unique task identification, called `task_addr`, which serves as an entry point for that function in the middleware component on the host machine. In Figure 4, the user registers function `sample_func` as the processing function for data messages sent to subtask 1 of periodic task 1.

```

void rt_init_app(void)
{
    struct task_addr addr;
    void (*sample_func)(void *);
    ... ..

    addr.task_id = 1;
    addr.task_type = 'P';
    addr.subtask_id = 1;

    rt_func_reg(addr, sample_func);

    ... ..
}

```

Figure 4. The `rt_init_app()` function.

On the other hand, registration of a data processing function on a particular machine only implies the presence of the binary code of the function on that machine. Whether or not the function will process the data depends upon the resource

allocation decision. If the resource manager determines that a data message needs to be processed by r replicas, the workload carried by that data message is distributed among the r replicas. The functions corresponding to the replicas on different host machines are then invoked with part of the workload.

Each data processing function is encompassed by a pair of `request_cpu()/release_cpu()` function calls, as shown in Figure 5. These two functions allow the data processing function to interact with the underlying Meta Scheduler as a distinct scheduling entity. Once the input workload has been processed, the data processing function may further send the processed data to the next subtask by calling `send_to_dest()`. Similar to the function registration, `send_to_dest()` accepts a destination subtask identified by a `task_addr` data structure, and the outgoing workload. Internally, the RTSL implements a separate thread that is responsible for accepting the resource allocation decision from the resource manager. Since the RTSL is aware of the allocation decision, i.e. the number of replicas for each subtask and their host machine assignment, it can always locate the replicas and distribute the workload among the replicas, if the `send_to_dest()` call is invoked.

```

void sample_func(void *data)
{
    ... ..
    request_cpu(...);

    /* process the data */
    ... ..

    /* send processed data to the next subtask */
    send_to_dest(...);

    /* notify the RM */
    send_to_monitor(...);

    release_cpu(...);
}

```

Figure 5. A sample processing function.

Furthermore, recall that the outgoing packets are scheduled by our real-time packet scheduling algorithms, e.g. BPA, on the network device driver as well as on the Real-Time Switch. To use the packet scheduling algorithms, proper real-time attributes should be passed to the network device driver on the host machines. This is implemented inside the `send_to_dest()` and hence is also transparent to the user.

Besides sending data to the next subtask, the user function may also notify the resource manager of start and finish of an end-to-end task, or even of a subtask, by calling `send_to_monitor()`. A location service component inside the RTSL automatically determines the location of the resource

manager and sends performance monitoring messages to it. Accordingly, a separate thread inside the resource manager can record the system performance and may react to possible timing failures.

7 The Resource Manager

The current implementation of the Resource Manager contains the RBA* and OBA resource allocation algorithms. For task scheduling, RBA* analyzes the response times under the DASA [3] and the BPA [14] algorithms. The scheduling algorithms themselves are implemented inside the Meta Scheduler. For packet scheduling, RBA* considers non-preemptive versions of DASA and BPA, on the host network driver as well as on the Real-Time Switch.

The Resource Manager is triggered at two events: a new user input of system configuration (number of host machines, etc.), adaptation functions, task structures and task timing requirements through the IDME; and a change of system resources detected by the Failure Detector(FD). In the former situation, the user simply specifies the system architecture, e.g. number of tasks, task structures and their timing requirements, number of host machines, etc., and then manually activates the resource manager in the IDME. We discuss features of the IDME in Section 8.

To detect the change of system resources, i.e. the available host machines, the *Choir* middleware executes a Failure Detector on the same machine as the resource manager. Additionally, each host machine in the system runs a Failure Detector Agent(FDA) component. The FDA continually sends numbered heartbeat messages to the FD at fixed rate. Thus, the FD can reason about the liveness of the host machines by examining the heartbeat messages. The current *Choir* implementation uses a simplified version of the failure detection algorithm proposed by Chen, Toueg, et al. [2]. Note that besides detecting host failures, the FD component can also discover the inclusion of new host machines in the system. Whenever a host machine joins the system, its FDA automatically locates the FD as well as the resource manager. After that, the FDA component begins to send heartbeat messages to the FD. Therefore, the FD is essentially informed the existence of a new host machine. Furthermore, once a host failure is detected, the failed host is permanently removed from the alive host list. Its recovery will be viewed as an event of a new host joining the system.

Once the resource manager is triggered, the RM executes the resource allocation algorithm to determine the number of replicas and their processor assignment for each subtask of the applica-

tion tasks, based on the latest information regarding task requirements, available system resources, etc. Since the user may not change the adaptation function for a long time, e.g. several hours if he or she does not expect the workload pattern will change during that future time interval, the RM will re-allocate the resources periodically. This is because the resource allocation is done only for a finite number of periods into the future. Allocation large number of periods imposes significant run-time overhead and hence is useless in practice. Furthermore, any timing analysis error will be amplified when the number of allocation periods increases. Once a allocation is finished, the RM broadcasts the new allocation decision on the network. The new allocation decision will be received by a thread inside the RTSL. In general, all host machines have consistent view of the allocation results. Note that the current middleware implementation focuses on proof of the concept. Thus, features such as reliable broadcast are not included in the current version.

Besides allocating resources, the RM also combines the functionality of performance monitoring for possible reaction to run-time timing failures. Recall that performance monitoring is done by the user code calling RTSL function `send_to_dest()`. Thus, the monitoring messages are sent to the RM, which in turn sends selected performance metrics to the IDME for visualization.

8 Integrated Development and Monitoring Environment (IDME)

For convenience, we have developed an Integrated Development and Monitoring Environment (IDME). The IDME serves three purposes:

1. Graphically composing and describing the system;

Ideally, the IDME allows the user to compose the system using graphical icons. Timeliness and other requirements can also be specified graphically. We have developed functionalities that support the graphic specification of the system. Once the IDME gets all system information, including the user data processing functions, it can automatically generate the remaining code using the *Choir* APIs, and hence construct the whole software system.

2. Visualizing the algorithm outputs;

The output of the algorithms are basically the number of replicas and their host machine assignment for each subtask. The current IDME has the capability of graphically

displaying the network topology, task structure, and the algorithm output, i.e. replica assignment.

3. Visualizing the system performance.

The IDME accepts selected performance metrics calculated by the resource manager and visualizes the system performance and status in an intuitive way. In Section 9, we use snapshots of the performance monitoring window during the real experiments as examples. Besides visualizing the system performance, the IDME can also display status of the system, e.g. liveness of the host machines in the system.

9 Experimental Results

We conducted experiments to study the effectiveness of the resource allocation algorithms and the *Choir* implementation. The experimental system consists of four end-host machines and one separate machine running the resource manager and the IDME. All five machines are connected to the Real-Time Switch through dedicated 100 Mbps full-duplex fast Ethernet links. The clocks of the end-host machines are synchronized using NTP 4.1.1 [11].

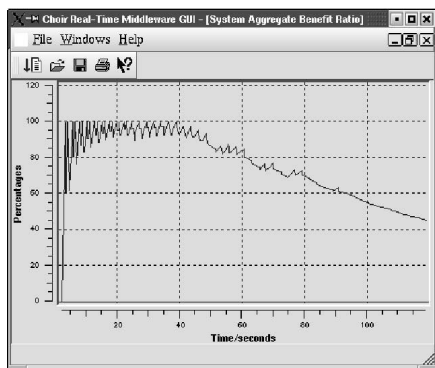


Figure 6. System AGB.

The initial experiments use the DASA algorithm for task scheduling. To be consistent, we also use the non-preemptive version of DASA to schedule packets on the end hosts and on the Real-Time Switch. The Resource Manager runs the RBA* algorithm to allocate resources for each sub-task of tasks. Two end-to-end tasks, T_0 and T_1 , are initiated in the system. The parameters of the two tasks are intentionally chosen to be exactly the same, except for the task benefits. Thus, we expect that the higher benefit task (T_0) may accrue more benefit than the lower benefit task

(T_1) does during overloaded situations. Each end-to-end task has five subtasks. The relative deadlines of a periodic task and its corresponding aperiodic task are set to 2.33 sec and 3.2 sec, respectively. We chose the “ramp/ramp” workload pattern, where the workload feed to a periodic task starts with 10 data objects/period and increases by 1 data object every five periods. In the meanwhile, the aperiodic workload is set at 80% of the workload of its triggering periodic task.

Figure 6 and Figure 7 show the measured system-level Aggregate Benefit ratio (AGB) and Deadline Satisfaction Ratio (DSR)¹. Initially, when the workload is not so heavy (before time 40 sec), the resource allocation results can satisfy the timing requirements of both tasks. Thus, we see that the two ratios are very close to 100%. However, when the load increases, the system is no longer able to satisfy the requirements of all tasks. Therefore, the system performance in terms of AGB and DSR begin to drop at around time 40 sec. Interestingly, the system AGB is always slightly higher than the DSR, as RBA* and the underlying scheduling algorithms seek to maximize the aggregate benefit, not the number of satisfied task deadlines.

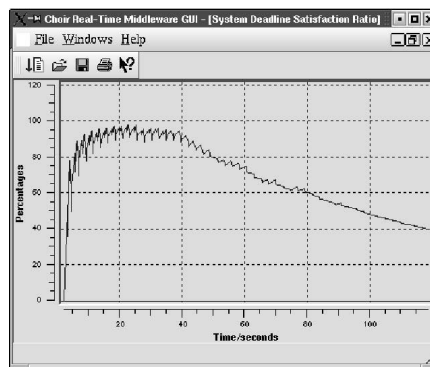


Figure 7. System DSR.

Furthermore, we observe that task T_0 accrues more benefit than task T_1 during overloaded situations. Figure 8 and Figure 9 show the aggregate benefit ratios of periodic tasks T_0 and T_1 , respectively. Task T_1 begins to miss its deadlines and thus suffers loss of benefit before time 40 sec. On the contrary, task T_0 reaches that stage after time 50 sec. Furthermore, the performance of task T_0 degrades more gracefully than that of task T_1 . Thus, these initial experimental results demonstrate the effectiveness of the RBA* algorithm and the *Choir* implementation.

¹These figures are snapshots of the *Choir* performance monitoring GUI windows.

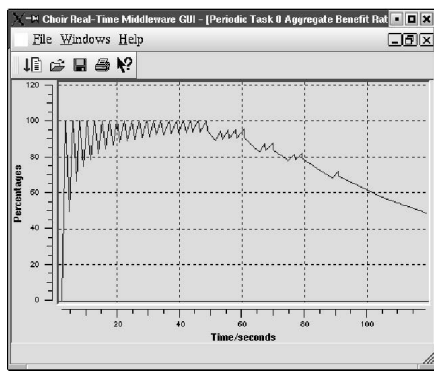


Figure 8. T_0 Aggregate Benefit Ratio.

10 Conclusions and Future Work

This paper describes a middleware implementation of our *proactive* resource allocation algorithms, called *Choir*. The *Choir* middleware system allows the user express end-to-end timeliness using Jensen's benefit functions, and transparently replicates (possibly migrates) the application subtasks, to conquer the uncertainties such that the system aggregate benefit is maximized. We showed effectiveness of the *Choir* middleware system through experiments.

Several aspects of the next generation *Choir* system are being developed. For example, we are currently working on implementing the algorithms inside real-time CORBA 2.0 dynamic scheduling framework [4].

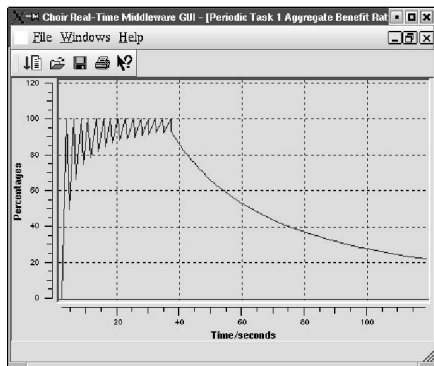


Figure 9. T_1 Aggregate Benefit Ratio.

Acknowledgements

This work was supported by the US Office of Naval Research under Grants N00014-99-1-0158 and N00014-00-1-0549.

References

- [1] T. Abdelzaher, S. Dawson, W.-C. Feng, et al. Armada middleware suite. In *Proceedings of The IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, pages 11–18, December 1997.
- [2] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, May 2002.
- [3] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990. CMU-CS-90-155.
- [4] O. M. Group. Dynamic scheduling real-time corba 2.0 (joint revised submission), 2001. orbos/2001-04-01 ed.
- [5] T. Hegazy and B. Ravindran. On decentralized proactive resource allocation in asynchronous real-time distributed systems. In *Proceedings of IEEE/IEICE International Symposium on High Assurance Systems Engineering*, pages 27–34, October 2002.
- [6] T. Hegazy and B. Ravindran. Using application benefit for proactive resource allocation in asynchronous real-time distributed system. *IEEE Transactions on Computers*, 51(8):945–962, August 2002.
- [7] E. D. Jensen. Asynchronous decentralized real-time computer systems. In W. A. Halang and A. D. Stoyenko, editors, *Real-Time Computing*. Springer Verlag, October 1992.
- [8] E. D. Jensen and B. Ravindran. Guest editor's introduction to special section on asynchronous real-time distributed systems. *IEEE Transactions on Computers, The IEEE Computer Society*, 51(8):881–882, August 2002.
- [9] P. Li and B. Ravindran. Proactive qos negotiation in asynchronous real-time distributed systems. *Journal of Systems and Software*. accepted December 2002.
- [10] P. Li, B. Ravindran, and S. Feizabadi. A framework for implementing real-time scheduling algorithms over posix-compliant operating systems. submitted to ACM Languages, Compilers, and Tools for Embedded Systems (LCTES'03), February 2003.
- [11] D. L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networks*, pages 245–254, June 1995.
- [12] B. Ravindran. Engineering dynamic real-time distributed systems: Architecture, system description language, and middleware. *IEEE Transactions on Software Engineering*, 28(1):30–57, January 2002.
- [13] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the tao real-time object request broker. *Computer Communication*, 21(4), April 1998.
- [14] J. Wang and B. Ravindran. Bpa: A fast packet scheduling algorithm for real-time switched ethernet networks. In *Proceedings of The 2002 IEEE International Conference on Parallel Processing (ICPP)*, pages 519–526, August 2002.