QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control *

Tarek F. Abdelzaher, Ella M. Atkins, and Kang G. Shin

Real-time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122
{*zaher, marbles, kgshin*}@*eecs.umich.edu*

Abstract

We propose a model for quality-of-service (QoS) negotiation in building real-time services to meet both predictability and graceful degradation requirements. QoS negotiation is shown to (i) outperform conventional "binary" admission control schemes (either guaranteeing the required QoS or rejecting the service request), and (ii) achieve higher application-perceived system utility. We incorporated the proposed QoS-negotiation model into an example real-time middleware service, called RTPOOL, which manages a distributed pool of shared computing resources (processors) to guarantee timeliness QoS for real-time applications. The efficacy and power of QoS negotiation are demonstrated for an automated flight control system implemented on a network of PCs running RTPOOL. This system is used to fly an F-16 fighter aircraft modeled using the Aerial Combat (ACM) F-16 Flight Simulator. Experimental results indicate that QoS negotiation, while maintaining real-time guarantees, enables graceful QoS degradation under conditions in which traditional schedulability analysis and admission control schemes fail.

1 Introduction

Predictability in real-time applications is often achieved by reserving resources and employing admission control under *a priori* assumed load and failure conditions. Graceful QoS degradation, on the other hand, requires dynamic resource reallocation in order to cope with changing load and failure conditions while maximizing system utility. Both predictability and graceful QoS degradation are necessary for real-time applications but pose conflicting requirements.

The main focus of this paper is on how to achieve predictability and graceful degradation in *long-lived* real-time services for embedded applications. By "long-lived" we mean that a request, if granted, will hold its reserved resources for a relatively long period of time. To control the load imposed on system resources and hence guarantee a certain level of QoS, the request must go through admission control and resource reservation. Conventional admission control schemes make "binary" decisions on whether to guarantee or reject each request. Future requests may be rejected because resources have already been committed to those that arrived earlier. In hard-real-time systems, a static analysis may be performed to guarantee *a priori* that all requests be honored under the assumption of the worst-case request arrival behavior and service requirements. If these assumptions are violated at run-time due to transient overload or resource loss (failures), the guarantees may become invalid, which may, in turn, lead to system failure.

We propose a mechanism for QoS (re)negotiation as a way to ensure graceful degradation in cases of overload, failures, or violation of pre-run-time assumptions. This mechanism permits clients to express in their service requests a *spectrum* of QoS levels they can accept from the provider and perceived utility of receiving service at each of these levels. As a result, the application designer will be able to express acceptable compromises in QoS and their relative cost/benefit as derived from application domain knowledge.

We incorporate the proposed QoS negotiation into a processing capacity management middleware service called RTPOOL. The service is designed and imple-

*The work reported in this paper was supported in part by the Advanced Research Projects Agency, monitored by the US Air Force Rome Laboratory under Grant F30602-95-1-0044.

mented to support timeliness guarantees for a flight control application, in which a set of flight control tasks, their QoS levels, and the corresponding rewards are provided by the flight *mission planner*, and can be renegotiated, if necessary, using RTPOOL's QoS-negotiation support. The mission planner was developed in the context of the Cooperative Intelligent Real-time Control Architecture (CIRCA) [1], which computes task execution tradeoffs from application domain knowledge and alters the mission plan as required during QoS negotiation.

In this paper, we begin with a review of related work (Section 2), followed by a description of the proposed QoS-negotiation model (Section 3). Next (Section 4), we describe RTPOOL, a distributed processing resource management service that follows the proposed QoS-negotiation model, highlighting the synergy between RTPOOL components and QoS-negotiation support. We present details of RTPOOL implementation and negotiation API (Section 5), then describe the use of RTPOOL in the context of automated flight control (Section 6). Flight performance is evaluated (Section 7), illustrating the efficacy of QoS-negotiation support, followed by a brief paper summary (Section 8).

2 Related work

Predictable performance of real-time services has traditionally been achieved using resource reservation and admission control. In hard real-time systems, sufficient resources are reserved *a priori* for the application. Off-line schedulability analysis (e.g., [2–4]) is used to verify that the reserved resources are sufficient for meeting all timing constraints. Such analysis requires that the worst-case load/failure conditions be known at design time. For some applications, the worst-case load and failure conditions may be difficult to know, thus a mechanism is needed to ensure graceful performance degradation when the load or failure hypotheses are violated.

On-line admission control has been used to guarantee predictability of services where request patterns are not known in advance, e.g., establishment requests of real-time channels [5]. This concept has also been applied to resource reservation for dynamically-arriving real-time tasks, e.g., the Spring Kernel [6] and Dreams real-time system [7]. A main concern of this approach is predictability. Run-time guarantees given to admitted requests are never revoked even if they result in rejecting subsequently-arriving more important requests competing for the same resources.

In soft real-time systems, services are more concerned with maximizing overall utility (by serving the most important request first) than guaranteeing reserved resources for individual requests. Priority driven services can generally be categorized this way, and are supported in real-time kernels such as Alpha [8] and Mach [9]. Under overload conditions, lower priority tasks are denied service in favor of more important tasks. In the Rialto operating system [10], a resource planner attempts to dynamically maximize user-perceived utility of the entire system. However, the scheme does not adopt the notion of guaranteeing a reserved amount of resources for the application.

Compromises between giving irrevocable service guarantees to arriving requests (in hard real-time system), and maximizing overall system utility (in soft real-time systems) have been addressed. For example, virtual clock based communication schemes [11] essentially delay a packet transmission request until its virtual arrival time. This enforces a global priority order, a special case of maximizing utility. Recently, a similar approach has been suggested for guaranteeing dynamic real-time tasks. The decision to guarantee an arrived task (and commit resources to it) is delayed until some critical instant, effectively making the system wait for "more important" tasks to arrive. Unfortunately, the delay in making task guarantees may itself waste processing bandwidth which may reduce schedulability and increase the task rejection rate.

A different approach to maintaining hard real-time guarantees while maximizing the overall perceived utility under overload and failure conditions is to offer QoS as a new dimension to trade in making resource management decisions. QoS negotiation extends the typical real-time service interface in two different ways. First, it offers QoS degradation as an alternative to denial of service, thus enhancing the percentage of accepting service requests and the total perceived system utility. Second, it provides a generic means of utilizing application-specific knowledge to control QoS degradation. This paper describes a generic QoS-negotiation scheme and its application to automated flight control systems.

3 QoS-Negotiation Model

A simple yet expressive QoS-negotiation model is the key to building predictable, gracefully degradable middleware services for real-time applications. In this section we describe the application model, the proposed QoS-negotiation model, and the model of a real-time middleware service that supports QoS negotiation. We consider a class of embedded real-time systems in which various software components realize functions to accomplish a single overall "mission." We will henceforth call this mission an *application*. Flight control, shipboard computing, automated manufacturing, and process control generally fall under this category. The application is composed of a set of tasks, each of which requires a set of resources/services. We are concerned mainly with long-lived services that need to hold reserved resources for an extended period of time, such as processor capacity reservation [12] and communication connection establishment services [5].

Our negotiation model is centered around three simple abstractions: *QoS levels*, *rewards*, and *rejection penalty*. A client requesting service specifies in its request a set of *negotiation options* and the penalty of rejecting the request derived from the expected utility of the requested service. Each negotiation option consists of an acceptable QoS level for the client to receive from the provider and a reward value commensurate with this QoS level. The QoS levels are expressed in terms of parameters whose semantics need to be known only to the client and service provider. For example, in establishing a real-time communication connection, these parameters may specify the client's traffic delay and jitter requirements. In processor capacity reservation, they may express the required processor bandwidth, while in a multicast protocol they may represent the semantics of the requested multicast service, such as reliable, ordered, causal, or atomic delivery. The reward represents the "degree of satisfaction" to be achieved from the QoS level (i.e., application-perceived utility of supplying the client with that level of service). Thus, the client's negotiation options represent a set of alternatives for "acceptable" QoS and their "utility." The *rejection penalty* of a client's request is the penalty incurred to the application if the request is rejected. Rejection penalty plays no further role if the request is guaranteed. In Section 6 we describe how QoS levels, negotiation options, and rejection penalty are computed in the context of a flight control application using a mission planner. The planner computes QoS levels, rewards, and penalties from application domain knowledge and a specification of system failure probabilities.

To control system load in a way that ensures predictable service, the service provider must subject the client's request to on-line admission control which determines whether to guarantee or reject the request. We propose a slightly different notion of guaranteeing a request, as compared to the conventional notion of guarantee. In our model, *guaranteeing* a client's request is the certification of the request to receive service at *one* of the QoS levels listed in its negotiation options. The selection of the QoS level it will actually receive, however, is up to the service provider. Furthermore, the service provider is free to switch this QoS level to another level in the client's negotiation options, if it increases perceived utility. Note that specifying only one negotiation option with default rejection penalty reduces this mechanism to traditional on-line guarantee schemes. Thus, while the proposed mechanism should perform no worse than these schemes in the special case, it provides the means to express and take advantage of more accurate semantic information about the application whenever such information is available. So, while we do *not require* the application designer to supply more information than is necessary for traditional on-line guarantee schemes, we *offer the flexibility* to take advantage of additional semantic information when it is available. In Section 6 we give an example application that benefits from this support.

Shifting the authority in selecting clients' QoS levels from client to service provider has two important advantages. First, the application code is decoupled from assumptions on underlying resource availability and capacity, implied when a client asks for a specific QoS level. Second, providing negotiation options and delegating QoS-level selection to the provider allows QoS-level adjustment by the provider, when necessary, to achieve higher overall system utility while maintaining each client's QoS guarantee at a level specified in negotiation options.

4 RTPOOL

We designed an example middleware service, RTPOOL, to support the proposed QoS-negotiation model. This service is responsible for managing a distributed pool of computing resources (processors) to guarantee timeliness. It employs a *processor membership protocol* to keep track of processor pool membership and report processor failures. Schedulability analysis is used to provide timeliness guarantees. Additionally, we integrated support for QoS negotiation into RTPOOL. This support is split into local and distributed algorithms, and is the focus of this section.

Clients of RTPOOL are application tasks. RTPOOL service requests are used to guarantee the timeliness of new incoming tasks. Our task execution model is influenced by the requirements of the flight control application (Section 6), but is still sufficiently general for use in different applications. RTPOOL assumes periodic tasks and handles aperiodic tasks with periodic servers. A task is composed of a set of modules and has a deadline by which all its modules must complete. The modules may have arbitrary precedence constraints among themselves, thus specifying their execution sequence. We assume task arrivals (guarantee requests) are independent, so we do not support precedence constraints among *different* tasks.

Each request for guaranteeing a task includes its rejection penalty, and the negotiation options of the client task that specify different QoS levels and their respective rewards. A client task's QoS level is specified by the parameters of its execution model. For an independent periodic task, the parameters consist of task period, deadline, and execution time. We model period and deadline as negotiable parameters. This represents a significant departure from most scheduling literature, although the authors of [13] articulate on the alterability of task periods in real-time control systems using system stability and performance index. Task execution time, on the other hand, depends on the underlying machine speed and thus should not be hardcoded into the client's request. Instead, each QoS level in the negotiation options specifies which modules of the client task are to be executed at that level. This allows the programmer to define different versions of the task to be executed at different QoS levels, or to compose tasks with mandatory and optional modules. The reward associated with each QoS level tells RTPOOL the utility of executing the specified modules of the task with the given period and deadline.

Requests for guaranteeing tasks may arrive dynamically at any machine in the pool. Since, in the proposed QoS-negotiation scheme, tasks normally receive higher QoS than their minimum functionality QoS level, it is highly probable for the new arrival to be guaranteed on the local machine. To guarantee a request at the local machine, RTPOOL executes a *local QoS-optimization heuristic*, which (re)computes the set of QoS levels for all local clients (including this new one) to maximize the sum of their rewards. Recomputing the QoS levels may involve degrading some tasks to accommodate the new one. The task is rejected if *both* (i) the new sum of rewards (including that of the newly-arrived task) is less than the existing sum prior to its arrival, *and* (ii) the difference between the current and previous sums is larger than the new task's rejection penalty. Otherwise, the requested task is guaranteed. As a result, task execution requests will be guaranteed unless the penalty from resulting QoS degradation of other local clients is larger than that from rejecting the request. When a task execution request is rejected by the local machine, one may attempt to transfer and guarantee it on a different machine using a load-sharing algorithm. Note that conventional admission control schemes would always incur the request rejection penalty whenever an

Let each client task T_i have QoS levels $M_i[0], \ldots, M_i[best_i]$ with rewards $R_i[0], \ldots, R_i[best_i]$, respectively.

1. Start by selecting the best QoS level, $M_i[best_i]$, for each client T_i.
2. While the set of selected QoS levels is *not* schedulable, do Steps 3 and 4.
3. For each client T_i receiving service at level $M_i[j] > M_i[0]$, determine the decrease of local reward, $R_i[j]-R_i[j-1]$, resulting from degrading this client to the next lower level.
4. Find client T_k whose $R_k[j]-R_k[j-1]$ is minimum and degrade it to the next lower level.
5. Go to Step 2.

Figure 1: Local QoS optimization heuristic

arrived task makes the set of current tasks unschedulable. By offering QoS degradation as an alternative to rejection and by using admission control rules, we can show that the reward sum (or perceived utility) achieved with our scheme is *lower bounded* by that achieved using conventional admission control schemes given the same schedulability analysis and load sharing algorithms. Thus, in general, our proposed scheme achieves higher perceived utility.

Figure 1 gives an example of the local QoS-optimization heuristic. The heuristic implements a gradient descent algorithm, terminating when it finds a set of QoS levels that keeps all tasks schedulable, if any. Note that unless all tasks are executed at their highest QoS level, the machine suffers from *unfulfilled potential reward*. The unfulfilled potential reward, UPR_j, on machine N_j, is the difference between the total reward achieved by the current QoS levels selected and the maximum possible reward that would be achieved if all local tasks were executed at their highest QoS level. This difference can be thought of as a fractional loss to the mission and is often unavoidable due to resource limitations. However, such loss may also be caused by poor load distribution, in which case it can be improved by proper load sharing.

RTPOOL employs a load-sharing algorithm that implements a *distributed QoS-optimization protocol*. Described in Figure 2, the protocol uses a hill climbing approach to maximize the global sum of rewards across all clients in the distributed pool. It is activated between two machines N_i and N_j when the difference $UPR_i - UPR_j$ exceeds a threshold V.

Close examination of the local QoS optimization

```
1. On source machine, $N_i$, find client $T_k$
   whose removal will result in maximum
   increase, $W$, in total reward.
2. $N_i$ request reassigning $T_k$ with reward
   $W$.
3. Each machine
   $N_j$, where $UPR_i - UPR_j > V$, receives
   the request and recomputes QoS levels
   for its local clients plus $T_k$. If its
   total reward is higher with $T_k$, $N_j$
   bids for $T_k$ with the reward increment
   $W_j$ resulting from accepting it.
4. $N_i$ transfers $T_k$ to highest bidder.
```

Figure 2: Distributed QoS optimization protocol

heuristic and the distributed QoS optimization protocol reveals that neither makes assumptions about the nature of the client and the semantics of its QoS levels.[1] For RTPOOL this means complete independence between the task model used by the feasibility assessment module and the QoS-negotiation mechanism. As a result, it is easier to enhance RTPOOL to handle more elaborate task models, constraints, and QoS-level parameters/semantics without affecting its QoS-negotiation mechanism. The disadvantage of this separation of concerns compromises optimality somewhat, as illustrated by the example in Section 7.

5 Implementation and API

In this section we highlight implementation details of RTPOOL, particularly those related to its QoS-negotiation API. RTPOOL is currently running on top of OSF Mach RT, mk7.2, on a PC platform, and is implemented as a user-level library which exports the abstraction of tasks, threads, QoS levels, and rewards. Highlighted below are the components of the implemented prototype.

5.1 Scheduling and QoS Negotiation

Our scheduling and QoS negotiation support is implemented as a thread package called *qthreads* controlled by a user-level local scheduler. The local scheduler is the lowest layer of RTPOOL, and supports *periodic thread* creation with a period that can be changed at run-time in response to QoS-level changes.

On top of our local scheduler, the *qthreads* package exports the abstraction of tasks, QoS levels, and rewards. Its API permits the user to create tasks, create threads within each task, define QoS levels, and specify rewards. It also permits the user to specify for a given thread the QoS levels in which the thread is eligible to execute. This package also exports a *force_negotiation()* primitive used to initiate QoS negotiation. In the current implementation, all created tasks execute in the same address space. The application is compiled into a single executable image that is loaded in its entirety at system start time. The code itself is thus static, although task arrival/activation times at different nodes may vary dynamically.

5.2 Invocation Migration

On top of *qthreads* we provide an invocation migration mechanism to implement the distributed QoS optimization protocol described in Section 4. The mechanism is completely transparent to the application. We call it *invocation* migration, because the transfer occurs between two successive invocations of a periodic task (i.e., when one invocation has terminated and the next hasn't started yet).

When the distributed QoS optimization heuristic determines that a task is to be migrated, the *state variables* of each thread in the transferred task are sent to the new machine, and the threads belonging to the task are destroyed at the source and recreated with the transferred state at the target. In the current implementation, state variables of a thread must be indicated to RTPOOL using a corresponding library call at thread initialization time. The *force_negotiation()* primitive is called on source and target after the transfer to update QoS levels accordingly. If a task must execute on a certain machine the task can be *wired* to that machine by calling a *wire_task()* primitive.

5.3 Pool Membership API

A membership algorithm is used to maintain a consistent view of current membership in the shared resource pool. Our group membership algorithm is a derivative of [14], and the user interface to this algorithm is the *subscribe_to_pool()* call which causes the machine on which the call was executed to join the named pool. When a new machine subscribes (joins), each machine in the pool adds the new member to the group. Since the new machine does not run any application task, its unfulfilled potential reward is zero. In our load-sharing heuristic, machines whose unfulfilled potential reward is above a given threshold will attempt to offload tasks to the new member. Task transfer will continue until the unfulfilled potential reward is balanced within a certain threshold, which stops the distributed QoS optimization protocol. When a machine fails, the group leader (the machine with the highest number in the pool) re-creates the destroyed tasks, then the load-sharing heuristic redistributes the load if necessary. When the group leader fails, its succes-

[1] The distributed QoS-negotiation protocol, however, assumes service to a given client can be migrated to another node.

sor (the machine with the next highest pool number) becomes the leader. Task state will be lost in case of a crash, but it can be avoided by task replication.

5.4 Communication API

An application need not be aware where each of its tasks is executing. The same executable application image is started on every machine that joins the pool. The application is composed of tasks, and the decision of where to run each task is left up to the load-sharing heuristic. This requires location-independent *send()* and *receive()* primitives for inter-task communication. Tasks communicate via location-independent *send()* and *receive()* primitives which use local communication buffers on the same machine, and send messages across the network for remote destinations. Our communication protocol stack is implemented using xKernel 3.2 [15], and is layered on top of a UDP/IP stack. The communication subsystem architecture on each host is designed to support prioritized, bounded-time message delivery. This architecture has been proposed earlier in the context of implementing real-time channels [16]. We adapt it to export the abstraction of a sporadic communication server, implemented as a separate task using *qthread* support.

6 Application

We have used RTPOOL to provide negotiable timeliness guarantees for several real-time tasks required in our fully-automated flight control system, which was used to fly a simulated model of an F-16 fighter aircraft. Details of the automated aircraft flight problem are provided in Section 6.1, followed by a description of a method to determine task QoS levels and rewards from application domain knowledge (Section 6.2). Section 6.3 summarizes the set of tasks, QoS levels, and rewards that describe our application.

6.1 Automated Flight Control

Current Flight Management Systems (FMS) ([17], [18]) perform several functions, including flight planning, navigation, guidance, and control. The flight planner computes waypoint trajectories, then during flight, the navigator uses sensor data to maintain a current aircraft state estimate. The guidance module uses the planned trajectory and state estimate to build the reference state vector, which is then used by the controller to compute actuator commands. In all FMS, real-time execution guarantees exist for navigation, guidance, and control, adhering to critical function deadlines with schedulability guarantees made off-line. Our QoS-negotiation scheme will allow graceful performance degradation when enough resources are lost to violate off-line guarantees. In this paper we consider only tasks having a known and bounded execution time. Issues in dealing with potentially unbounded on-line computations, such as run-time intelligent flight planning, are discussed in [1].

We issue F-16 aircraft guidance commands in terms of altitude (z) and compass heading (h), and employ a control loop to compute *primary* actuator commands, including elevator, ailerons, rudder, and throttle. The elevator, ailerons, and rudder generate aerodynamic forces that directly affect aircraft roll and pitch attitude, and, via dynamic coupling, alter aircraft heading and airspeed. The throttle provides a linear force along the aircraft fuselage. Our controller may also command a *secondary* actuator set that improves flight performance but is not critical for flight safety. Secondary actuators include the afterburner for extra engine thrust, and flaps and speed brake used to enhance slow-airspeed control.

In a parallel research effort [19] a set of linear controllers have been implemented to calculate the *primary* actuator commands to achieve the desired reference altitude (z_{ref}) and heading (h_{ref}). Controller state includes altitude z, heading h, pitch, p, and roll r. Equation (1) shows the control laws used during our experiments, adopted from those used in [1]. In higher-performance QoS levels (see Section 6.3), the controller also sends discrete-valued commands to the *secondary* actuator set (described in [19]).

$$\begin{pmatrix} elevator \\ ailerons \\ rudder \end{pmatrix} = K \begin{pmatrix} z_{ref} - z \\ h_{ref} - h \\ p \\ \dot{p} \\ r \\ \dot{r} \end{pmatrix} \quad (1)$$

$$K = \begin{pmatrix} K_1 & 0 & -K_{p_1} & -K_{d_1} & 0 & 0 \\ 0 & K_2 & 0 & 0 & -K_{p_2} & -K_{d_2} \\ 0 & K_3 & 0 & 0 & -K_{p_3} & -K_{d_3} \end{pmatrix}$$

6.2 Computing QoS Levels and Rewards

Our QoS-negotiation scheme enables the application domain expert to express application-level semantics to RTPOOL using QoS levels, rewards, and rejection penalty. In this section we briefly highlight how this support may complement mission planning techniques in the context of CIRCA (the Cooperative Intelligent Real-time Control Architecture) [1]. Based on a user-specified domain knowledge base, CIRCA's main goal is to build a set of control plans to keep the system "safe" (i.e., avoid catastrophic failures such as an aircraft crash) while working to achieve its performance goals (e.g., arrive at its destination on time).

In order to deal successfully with an inherently nondeterministic, perhaps poorly modeled, environment of a complex real-time system CIRCA employs *probabilistic* planning which models the system by a set of states and transition probabilities. System failure is modeled by temporal transitions to failure states (TTFs). CIRCA's mission planner uses its domain knowledge base to select appropriate actions (tasks) and their timing constraints (QoS levels), so that the probability of TTFs is reduced below a certain threshold. The reward decrease corresponding to degrading a task from one QoS level to another, or rejecting a task altogether, is computed from the corresponding increase in failure probability.

The CIRCA planner computes a maximum period for each task based on the notion of preempting TTFs [19]. For any state, an outgoing TTF is considered *preempted* if its probability is below the specified threshold. To define alternative QoS levels, CIRCA's planner may compute different task periods based on a set of alternative TTF probability thresholds. For example, say a TTF has a cumulative probability distribution that reaches the threshold value when the preemptive task's maximum period is set to 0.2 seconds. But, suppose we need to relax the task's period requirement under overload. The new, longer task period for degraded QoS is computed from the next higher TTF probability threshold, and this task is assigned a lower reward that corresponds to the reduction in certainty that the TTF will be preempted. A complete set of task QoS levels may be developed by considering all TTF probability thresholds.

6.3 Description of Flight Tasks

We have used the Aerial Combat (ACM) F-16 flight simulator [20] for all flight tests. ACM runs on a Sun workstation with socket connection to the real-time execution platform. We have tested QoS-negotiation by flying the simulated aircraft around a left-hand pattern in which the aircraft executes a takeoff and climb, holds a constant altitude around a rectangular course, then descends through final approach and landing. By varying periods for controllers and sensors, we are able to observe the degradation in flight quality (i.e., stability) as a function of each task's selected QoS level.

In this section, we describe the tasks and rewards used during our QoS-negotiation algorithm tests. Our mission goals were to complete the flight around the pattern and to destroy observed enemy targets, if any, using the F-16's onboard radar and missiles. Four separate tasks were required to control the aircraft: Guidance (*Guid*), Control (*Ctrl*), Slow Navigation (*SNav*), and Fast Navigation (*FNav*). These tasks function much like their similarly-named FMS counterparts. *Guid* is responsible for computing the reference trajectory (state) for the aircraft. In our tests, *Guid* specified heading and altitude to lead the aircraft around the pattern through landing. The *Ctrl* task is responsible for executing the low-level control loops to compute actuator commands from the high-level guidance trajectory. We have two navigation tasks (*SNav*,*FNav*) to estimate aircraft state, distinguished by required update frequency.

Table 1 shows the QoS levels (L) present for all tasks, including associated reward (R), execution time (ET), period (P), and version (Ver). In our simple tests, we set each task deadline equal to its period, although there are no such requirements in our QoS negotiation protocol. Also, because all tasks are considered critical to execute (at least at a degraded QoS level), we set all task rejection penalties sufficiently high that all tasks are always accepted by the QoS negotiator.

In addition to the basic flight control tasks, we simulate a function necessary during military operation: Missile Control (*MC*). *MC* is composed of two precedence-constrained threads: "Read Radar" and "Fire Missile". "Read Radar" monitors aircraft radar to detect approaching enemy targets, then "Fire Missile" launches a missile at any enemy targets appearing on radar. As shown in Table 1, *MC* is computationally expensive and has two QoS levels. In Level 1, radar will be scanned with sufficient frequency to allow detection and destruction of most enemy targets. Otherwise (Level 0), fast-moving targets may not be destroyed. During experiments (in Section 7.3), we varied the reward for *MC* QoS Level 1 depending on the relative importance of destroying enemy targets.

As described above, *Ctrl* is responsible for executing the control loop. At each invocation, *Ctrl* uses the Equation (1) control law with appropriate gains to compute *primary* actuator values. Two versions of this task were tested, one that used the *secondary* actuators (QoS levels 0, 2, and 4) and one that did not (QoS levels 1 and 3). Use of these actuators allows the aircraft to perform better in terms of takeoff distance and climb rate as shown in Section 7 at the expense of a longer task execution time. The importance of *Ctrl* task period is illustrated with relatively high reward given to low-period *Ctrl* QoS levels. The small reward changes between the use of the different versions (e.g., level 3 vs. level 4) reflects the fact that version choice is not critical for safety.[2]

[2]We defined a QoS "level 0" for *Ctrl* and *SNav* that, as will be shown in Section 7, were so slow that the aircraft becomes

Task	L	R	ET(ms)	P(sec)	Ver
Guid	0	10	100	10	def
	1	15	100	5	def
	2	20	100	1	def
Ctrl	0	1	80	5	sec
	1	100	60	1	prim
	2	104	80	1	sec
	3	120	60	0.2	prim
	4	124	80	0.2	sec
SNav	0	10	100	10	def
	1	20	100	5	def
	2	25	100	1	def
FNav	0	1	60	5	def
	1	100	60	1	def
	2	120	60	0.2	def
MC	0	1	500	10	def
	1	30/200	500	1	def

Table 1: Flight plan with different QoS levels.

The *SNav* task is responsible for reading sensors that do not require a high sampling rate. All navigation sensors are grouped into this task because they are used only by the *Guid* task to determine high-level altitude and heading commands. The reward/period values for *SNav* in Table 1 reflect this task's non-critical nature. Finally, *FNav* is responsible for maintaining all sensor data used by the *Ctrl* task. Since the system must read this data frequently to maintain sufficient state estimates, the periods and associated rewards are similar to those used by *Ctrl*.

7 Evaluation

In this section, we show results illustrating how QoS negotiation can help aircraft flight control degrade gracefully. First, we assess the QoS negotiation heuristic for our flight tasks by observing how the QoS of each task degrades with lower machine speeds. In Section 7.2 we study aircraft performance during flight as a function of the *Ctrl* task's QoS level, illustrating graceful performance degradation by example. In Sections 7.1 and 7.2, we focus on tests that use a single machine, and consider only the guidance, navigation, and control tasks. We conclude our experiments (Section 7.3) with tests which also include the missile control task and observe the effects of load sharing between two machines, with processor failure used to demonstrate graceful performance degradation.

unstable. These levels are included among the QoS negotiation options for illustrative purposes only.

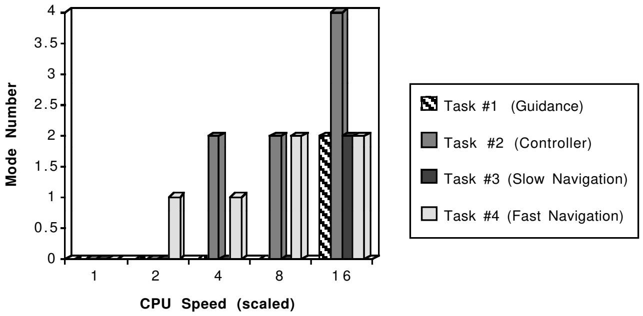

Figure 3: QoS levels vs. CPU speed for flight tasks.

7.1 QoS Negotiation Heuristic Testing

In Section 4, we described a simple local QoS optimization heuristic to help a service provider select a high-reward set of QoS levels for its clients. Using the QoS levels and rewards listed in Table 1, we illustrate the behavior of the presented heuristic. In this experiment we kept the task set fixed, and decreased the underlying CPU speed (increasing task execution times), then observed the corresponding decrease in task QoS levels. Figure 3 plots QoS levels (modes) selected vs. CPU speed, normalized by the minimum CPU speed for which the task set is schedulable.

Since the heuristic uses only reward information to guide its search for a feasible QoS level set (thus being applicable as-is in any service that uses our QoS negotiation scheme), optimality is compromised yet "graceful QoS degradation" is still illustrated.

7.2 Aircraft Performance

We evaluated system performance by studying its ability to control the simulated F-16 during flight. In this section, all flight control tasks execute on one machine. As shown in Figure 3, since *Ctrl* and *Fnav* required the smallest period, these tasks are the execution bottlenecks, so flight performance changes are most easily observed via changes in *Ctrl* or *Fnav* QoS. Since these tasks are tightly coupled (i.e., *Ctrl* uses results from *FNav*), our test matrix included variations of *Ctrl* QoS from its highest (4) to lowest (0) level and ensured that *FNav* acted with at least as low a period as that selected for *Ctrl*.

As shown in Table 1, *Ctrl* QoS levels are a function of both period and version. We present tests that show flight performance differences due to each of these variables, specifically during the critical takeoff/climb phase of flight. Figure 4 illustrates differences between the two versions of *Ctrl* in their "best performance" case (period = 200 msec). Level 4 (with *secondary* actuation) requires larger execution time than level 3 (no *secondary* actuation), thus is harder to schedule.

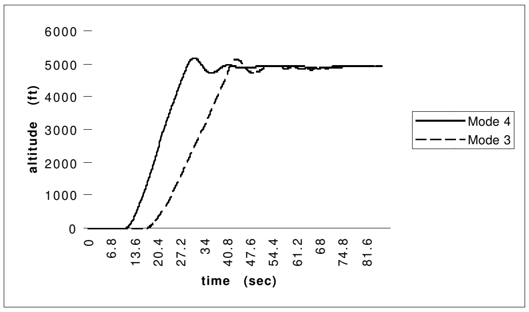

Figure 4: Aircraft altitude performance with and without *secondary* control actuation.

Climb performance for level 4 is only slightly better for level 3, consistent with their small reward difference. This example illustrates how QoS negotiation can achieve graceful degradation. Overall processor utilization is decreased by reducing *Ctrl* to level 3, but safety (controller stability) is not compromised.

Next, we performed tests with varying *Ctrl* task period. We isolated version from period effects by exclusively selecting QoS levels with *secondary* actuation, although similar trends result with the other *Ctrl* version (levels 1 and 3). To illustrate performance changes as a function of task period, we consider three QoS levels: level 4 (period = 0.2 sec), level 2 (period = 1 sec), and level 0 (period = 5 sec). We include level 0 among *Ctrl* QoS negotation options as a comparative example illustrating controller instability. Of course, no unstable QoS levels should be defined among a client's negotiation options, since the client should not "ask" for instability.

Figures 5 through 8 show aircraft state as a function of time during takeoff, climb, and a turn from South to East. Figure 5 shows aircraft altitude for the different *Ctrl* task periods. As period increases, climb performance gracefully degrades between QoS levels 4 and 2, but then becomes unstable in level 0 (period = 5 sec), illustrating the necessity of real-time response for the *Ctrl* task. Figures 7 and 8 show aircraft pitch angle and roll angle, respectively, for the "stable" controller QoS levels. Note that we do not include level 0 because the unstable response obscures the other plots. Pitch angle and altitude are coupled, so pitch has largest magnitude during the climb, and as illustrated, the period increase to 1 second causes a large pitch angle to be required longer, a stable but undesirable performance trait. Roll angle (Figure 8) shows delay and longer deviation from zero as well as significant overshoot when task period increases.

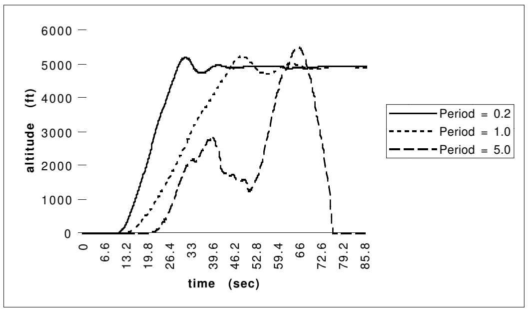

Figure 5: Aircraft altitude for varied *Ctrl* QoS levels.

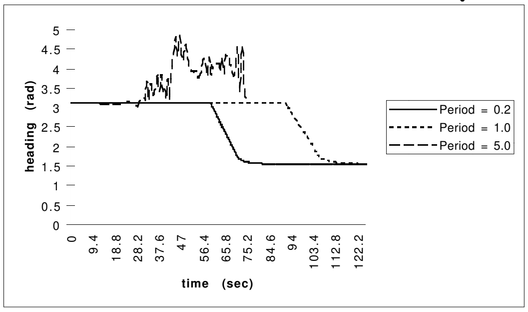

Figure 6: Aircraft heading for varied *Ctrl* QoS levels.

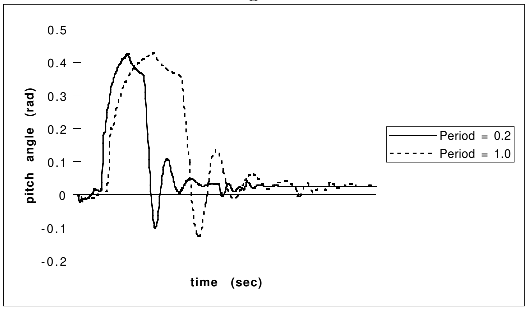

Figure 7: Aircraft pitch for varied *Ctrl* QoS levels.

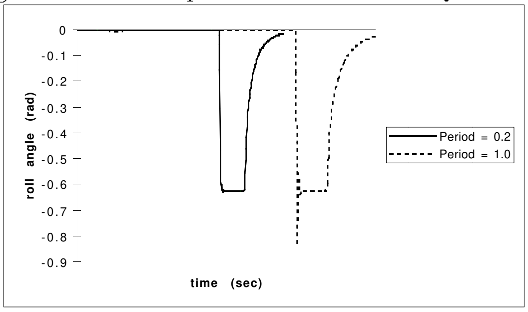

Figure 8: Aircraft roll for varied *Ctrl* QoS levels.

7.3 Load Sharing

Load sharing capabilities are implemented in RTPOOL, and we performed a final test set which included both the flight control tasks and the missile control *MC* task, as described in Section 6.3. In these tests, we start the system with two machines available for task execution. Because, as defined in Table 1, the *MC* task is computationally expensive, the load sharing protocol places all flight control tasks on one machine and the *MC* task (both "Read Radar" and "Fire Missile" threads) on the other machine.

When the two machines function normally, both flight and missile control tasks ran in their maximum performance levels. In this case, enemy targets are quickly detected and fired upon, while flight control is identical to the best performance profiles in the Section 7.2 plots. For the next test set, we began operation with two functioning machines, then shut one down (simulating failure) just after takeoff. This requires the load sharing and QoS negotiation algorithms to function dynamically, such that the one functional machine now has to execute both flight and missile control tasks. If *MC* task reward is relatively low, the system chooses to degrade the *MC*, *Guid*, and *SNav* functions (to level 0), but manages to keep the *Ctrl* and *FNav* tasks at safe levels. In this manner, flight control is a bit sluggish but stable. However, the aircraft is unable to launch missiles at most targets.

Alternatively, this system may be aboard an expendable drone whose most important function is to destroy a target or attack enemy aircraft. In this case, the reward set may be structured such that the missile control task takes precedence over accurately maintaining flight control.[3] To illustrate such changes in the task reward set, we altered the reward for QoS level 1 of *MC* to 200 (shown in Table 1). Now, when the second machine shuts down, the QoS negotiator reduces all flight control levels to 0, since the missile controller is the "most important" task. After one machine fails, the aircraft eventually becomes unstable (as illustrated in Section 7.2), but will still quickly detect and respond to enemy targets on radar.

It is important to note that, had we used traditional schedulability analysis algorithms that do not allow negotiated QoS degradation, the system would have failed to guarantee/accept the entire task set on the same processor, leading to complete mission failure.

[3] In our tests, when missile control takes precedence over flight control during single machine operation, the aircraft becomes unstable. This would be undesirable in an actual system, since missiles cannot be launched from a crashed aircraft.

8 Summary and Future Work

In this paper we presented a novel scheme for QoS negotiation in real-time applications. This scheme is applicable for the design of real-time service providers, extending the interface of such services in that (i) it adopts a modified notion of request guarantees that allows for defining QoS compromises and supports graceful QoS degradation, and (ii) it provides a generic means to express application-level semantics to control how application QoS is to be degraded under overload or failure conditions. Our QoS-negotiation scheme improves the guarantee ratio over traditional admission control algorithms and increases the application-level perceived utility of the system.

The proposed QoS-negotiation architecture has been incorporated into RTPOOL, an example middleware service which implements a computing resource manager for a pool of processors. The synergy between components of the service and QoS-negotiation support has been illustrated. RTPOOL is used for a flight control application to demonstrate the efficacy of QoS negotiation. We demonstrated that the application does have negotiable parameters and can thus benefit from the added flexibility of negotiation. We also showed that application task QoS levels and their rewards can be analytically derived from system failure probability. QoS-negotiation support, while guaranteeing maximum QoS levels during normal operation, is shown to provide graceful QoS degradation in case of resource loss.

We have demonstrated how an application can benefit from the proposed QoS-negotiation scheme, but we have not analyzed the performance of different QoS optimization policies, nor the general scope of their applicability. We are currently studying alternative QoS-optimization methodologies and the scalability of our QoS-negotiation approach. We are also considering ways to implement negotiable fault-tolerance QoS, perhaps as an extension to RTPOOL. Finally, we are working to develop new schemes for quantifying perceived utility to compute reward and penalty values. Possible approaches include adapting performability analysis and using economic models of utility/costs.

Acknowledgment

The authors wish to thank Farnam Jahanian, Ashish Mehra, Anees Shaikh, and Wuchang Feng for sharing their opinions and insights during the development of this paper.

References

[1] E. M. Atkins, E. H. Durfee, and K. G. Shin, "Plan development in CIRCA using local probabilistic models," in *Uncertainty in Artificial Intelligence:*

Proceedings of the Twelfth Conference, pp. 49–56, August 1996.

[2] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. Software Engineering*, vol. SE-16, no. 3, pp. 360–369, March 1990.

[3] T. Shepard and M. Gagne, "A pre-run-time scheduling algorithm for hard real-time systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 669–677, Jul 1991.

[4] T. F. Abdelzaher and K. G. Shin, "Optimal combined task and message scheduling in distributed real-time systems," in *IEEE Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[5] 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.

[6] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A new paradigm for real-time systems," *IEEE Software*, pp. 62–72, May 1991.

[7] S. Sommer and J. Potter, "Operating system extensions for dynamic real-time applications," in *IEEE Real-Time Systems Symposium*, pp. 45–50, Washington, DC, December 1996.

[8] R. Clark, E. Jensen, and F. Reynolds, "An architectural overview of the Alpha real-time distributed kernel," in *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, 1992.

[9] H. Tokuda, T. Nakajima, and P. Rao, "Real-time Mach: Towards a predictable real-time system," in *Proceedings of the USENIX Mach Workshop*, pp. 73–82, October 1990.

[10] M. B. Jones and P. J. Leach, "Modular real-time resource management in the rialto operating system," Technical Report MSR-TR-95-16, Microsoft Research, Advanced Technology Division, May 1995.

[11] W. Zhao and K. Ramamritham, "Virtual time CSMA protocols for hard real-time communication," *IEEE Transactions of Software Engineering*, vol. 13, no. 8, pp. 938–952, 1987.

[12] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pp. 90–99, May 1994.

[13] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin, "On task schedulability in real-time control systems," in *IEEE Real-Time Systems Symposium*, pp. 13–21, Washington, DC, December 1996.

[14] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin, "RTCAST: Lightweight multicast for real-time process groups," in *IEEE Real-Time Technology and Applications Symposium*, Boston, Massachusetts, June 1996.

[15] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, January 1991.

[16] A. Mehra, A. Indiresan, and K. G. Shin, "Structuring communication for quality of service guarantees," in *IEEE Real-Time Systems Symposium*, pp. 144–154, Washington, DC, December 1996.

[17] S. Liden, "The evolution of flight management systems," in *Proceedings of the 1994 IEEE/AIAA Thirteenth Digital Avionics Systems Conference*, pp. 157–169. IEEE, 1995.

[18] J. Schreur, "B737 flight management computer flight plan trajectory computation and analysis," in *Proceedings of the American Control Conference*, pp. 3419–3429, June 1995.

[19] E. M. Atkins. *Reasoning About and In Time when Building Plans for Safe, Fully-Automated Aircraft Flight.* Ph.D. Thesis Proposal, December 1996.

[20] R. Rainey. *ACM: The Aerial Combat Simulation for X11*, February 1994.