

Proceedings

The 19th IEEE Real-Time Systems Symposium

December 2 - 4, 1998
Madrid, Spain

Sponsored by
IEEE Computer Society Technical Committee on Real-Time Systems
The Technical University of Madrid



Los Alamitos, California

Washington • Brussels • Tokyo

Copyright © 1998 by The Institute of Electrical and Electronics Engineers, Inc.
All rights reserved

Copyright and Reprint Permissions: Abstracting is permitted with credit to the source. Libraries may photocopy beyond the limits of US copyright law, for private use of patrons, those articles in this volume that carry a code at the bottom of the first page, provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Other copying, reprint, or republication requests should be addressed to: IEEE Copyrights Manager, IEEE Service Center, 445 Hoes Lane, P.O. Box 133, Piscataway, NJ 08855-1331.

The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and, in the interests of timely dissemination, are published as presented and without change. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society, or the Institute of Electrical and Electronics Engineers, Inc.

IEEE Catalog Number 98CB36279
ISBN: 0-8186-9212-X (Softbound)
ISBN: 0-7803-5243-2 (Casebound)
ISBN: 0-8186-9214-6 (Microfiche)
ISSN: 1052-8725
PR8212

Additional copies may be ordered from:

IEEE Computer Society Customer Service Center 10662 Los Vaqueros Circle P.O. Box 3014 Los Alamitos, CA 90720-1314 Tel: + 1-714-821-8380 Fax: + 1-714-821-4641 E-mail: cs.books@computer.org	IEEE Service Center 445 Hoes Lane P.O. Box 1331 Piscataway, NJ 08855-1331 Tel: + 1-732-981-1393 Fax: + 1-732-981-9667 mis.custserv@computer.org	IEEE Computer Society Asia/Pacific Office Watanabe Building, 1-4-2 Minami-Aoyama Minato-ku, Tokyo 107-0062 JAPAN Tel: + 81-3-3408-3118 Fax: + 81-3-3408-3553 tokyo.ofc@computer.org
--	---	--

Editorial production by Kristine Kelly

Cover art production by Joseph Daigle/Studio Productions

Printed in the United States of America by The Printing House



Table of Contents

Foreword	ix
Conference Organization	xi
Program Committee	xii
Reviewers	xiii

Session 1: Keynote Talk

Future Challenges in the Development of Real-Time High Performance Embedded Systems.....	1
<i>David Martinez, MIT Lincoln Laboratory</i>	

Session 2: Systems and Scheduling I

Integrating Multimedia Applications in Hard Real-Time Systems	4
<i>L. Abeni and G. Buttazzo</i>	
Isochronous Scheduling and its Application to Traffic Control.....	14
<i>M. Iwasaki, T. Takeuchi, M. Nakahara, and T. Nakano</i>	
Schedulability Analysis for Tasks with Static and Dynamic Offsets.....	26
<i>J.C. Palencia and M. González Harbour</i>	

Session 3.A: Databases and Filesystems

Deadline-Modification-SCAN with Maximum-Scannable-Groups for Multimedia Real-Time Disk Scheduling	40
<i>R.-I. Chang, W.-K. Shih, and R.-C. Chang</i>	
Using Separate Algorithms to Process Read-Only Transactions in Real-Time Systems	50
<i>K.-W. Lam, S.H. Son, V.C.S. Lee, and S.-L. Hung</i>	
Maintaining Temporal Coherency of Virtual Data Warehouses.....	60
<i>R. Srinivasan, C. Liang, and K. Ramamritham</i>	

Session 3.B: Systems Design and Development I

Specification and Modeling of Dynamic, Distributed Real-Time Systems	72
<i>L.R. Welch, B. Ravindran, B.A. Shirazi, and C. Bruggeman</i>	
Detailed Design of Avionics Control Software.....	82
<i>U. Nilsson, S. Streiffert, and A. Törne</i>	
Schedulability Analysis for Automated Implementations of Real-Time Object-Oriented Models	92
<i>M. Saksena, A. Ptak, P. Freedman, and P. Rodziewicz</i>	

Session 4.A: Scheduling and Analysis I

A Better Polynomial-Time Schedulability Test for Real-Time Multiframe Tasks	104
<i>C.-C.J. Han</i>	
A General Model for Recurring Real-Time Tasks	114
<i>S.K. Baruah</i>	
Statistical Rate Monotonic Scheduling	123
<i>A. Atlas and A. Bestavros</i>	

Session 4.B: Timing Analysis and Compiler Techniques

Testing the Results of Static Worst-Case Execution-Time Analysis	134
<i>P. Puschner and R. Nossal</i>	
Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis	144
<i>H. Theiling and C. Ferdinand</i>	
Compiler Optimizations for Real Time Execution of Loops on Limited Memory Embedded Systems.....	154
<i>S. Anantharaman and S. Pande</i>	

Session 5: Keynote Talk

Animated Structures: Real-Time, Objects, and the UML.....	165
<i>Bran Selic, ObjecTime, Ltd.</i>	

Session 6: Systems and Scheduling II

The Time-Triggered Model of Computation	168
<i>H. Kopetz</i>	
Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors.....	178
<i>I. Hong, G. Qu, M. Potkonjak, and M.B. Srivastava</i>	
Task Period Selection and Schedulability in Real-Time Systems.....	188
<i>D. Seto, J. P. Lehoczky, and L. Sha</i>	

Session 7.A: Testing, Verification and Analysis I

Automatic Testing of Reactive Systems	200
<i>P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber</i>	
Analyzing Non-Deterministic Real-Time Systems with (max,+) Algebra	210
<i>G.P. Brat and V.K. Garg</i>	
Timed Test Cases Generation Based on State Characterization Technique.....	220
<i>A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi</i>	

Session 7.B: Operating Systems and Services

General Data Streaming	232
<i>F.W. Miller, P. Keleher, and S.K. Tripathi</i>	
Design and Implementation of a Real-Time ATM-Based Protocol Server	242
<i>M. Borriss and H. Härtig</i>	
Proportional Share Scheduling of Operating System Services for Real-Time Applications	480
<i>K. Jeffay, F.D. Smith, A. Moorthy, and J. Anderson</i>	

Session 8.A: Testing, Verification and Analysis II	
Membership Questions for Timed and Hybrid Automata	254
<i>R. Alur, R.P. Kurshan, and M. Viswanathan</i>	
On Checking Timed Automata for Linear Duration Invariants	264
<i>V.A. Braberman and D.V. Hung</i>	
Schedulability Analysis of Acyclic Processes	274
<i>M.J. Meyer and H. Wong-Toi</i>	
Session 8.B: Quality of Service Issues	
Elastic Task Model for Adaptive Rate Control	286
<i>G.C. Buttazzo, G. Lipari, and L. Abeni</i>	
Practical Solutions for QoS-Based Resource Allocation.....	296
<i>R. Rajkumar, C. Lee, J.P. Lehoczy, and D.P. Siewiorek</i>	
A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage	307
<i>S. Brandt, G. Nutt, T. Berk, and J. Mankovich</i>	
Session 9: Keynote Talk	
From POSIX Threads to Ada to Java: A Brief History of Runtime Development for Some Real-Time Programming Languages.....	319
<i>Ted Baker, Florida State University</i>	
Session 10: Operating Systems and Runtimes	
Techniques for Software Thread Integration in Real-Time Embedded Systems.....	322
<i>A.G. Dean and J.P. Shen</i>	
A Worst Case Timing Analysis Technique for Multiple-Issue Machines	334
<i>S.-S. Lim, J.H. Han, J. Kim, and S.L. Min</i>	
Efficient Object Sharing in Quantum-Based Real-Time Systems	346
<i>J.H. Anderson, R. Jain, and K. Jeffay</i>	
Session 11.A: Systems Design and Development Tools II	
A Development Framework for Ultra-Dependable Automotive Systems Based on a Time-Triggered Architecture	358
<i>B. Hedenetz</i>	
Integrated Design Tools for Hard Real-Time Systems	368
<i>C. Puchol and A.K. Mok</i>	
Performance Measurement Using Low Perturbation and High Precision Hardware Assists	379
<i>A. Mink, W. Salamon, J. K. Hollingsworth, and R. Arunachalam</i>	
Session 11.B: Scheduling and Analysis II	
Real-Time Scheduling in a Generic Fault-Tolerant Architecture.....	390
<i>A.J. Wellings, L. Beus-Dukic, and D. Powell</i>	
Improved Response-Time Analysis Calculations	399
<i>M. Sjödin and H. Hansson</i>	
Symbolic Schedulability Analysis of Real-Time Systems	409
<i>H.-H. Kwak, I. Lee, A. Philippou, J.-Y. Choi, and O. Sokolsky</i>	

Session 12.A: Dependability and Fault Tolerance	
Fault-Tolerant Clock Synchronization in CAN	420
<i>L. Rodrigues, M. Guimarães, and J. Rufino</i>	
Using Light-Weight Groups to Handle Timing Failures in Quasi-Synchronous Systems	430
<i>C. Almeida and P. Veríssimo</i>	
Dependable System Upgrade.....	440
<i>L. Sha</i>	
Session 12.B: Communications and Networks	
Statistical Delay Guarantee of Virtual Clock.....	450
<i>P. Goyal and H.M. Vin</i>	
Realizing Services for Guaranteed-QoS Communication on a Microkernel Operating System.....	460
<i>A. Mehra, A. Shaikh, T. Abdelzaher, Z. Wang, and K.G. Shin</i>	
Scheduling Communication Networks Carrying Real-Time Traffic	470
<i>J.P. Lehoczky</i>	
Author Index	492

Specification and Modeling of Dynamic, Distributed Real-time Systems

Lonnie R. Welch, Binoy Ravindran, Behrooz A. Shirazi and Carl Bruggeman
Computer Science and Engineering Dept.
The University of Texas at Arlington
Box 19015, Arlington, TX 76019-0015
{welch|binoy|shirazi|bruggema}@cse.uta.edu

Abstract

Time-constrained systems which operate in dynamic environments may have unknown worst-case scenarios, may have large variances in the sizes of the data and event sets that they process (and thus, have large variances in execution latencies and resource requirements), and may not be statically characterizable, even by time-invariant statistical distributions. This paper presents a specification language for describing environment-dependent features. Also presented is an abstract model that is constructed (statically) from the specifications, and is augmented (dynamically) with the state of environment-dependent features. The model is used to define techniques for QoS (quality-of-service) monitoring, QoS diagnosis, and resource allocation analysis. Experimental results show the effectiveness of the approach for specification of real-time QoS, detection and diagnosis of QoS failures, and restoration of acceptable QoS via reallocation of distributed computer and network resources.

1. Introduction

Many difficulties confront those who must engineer the emerging generation of real-time control systems. Such systems have rigorous Quality of Service (QoS) objectives. They must behave in a dependable manner, must respond to threats in a timely fashion and must provide continuous availability, even within hazardous and unknown environments. Furthermore, resources should be utilized in an efficient manner, and scalability must be provided to address the ever-increasing complexity of scenarios that confront such systems. The difficulties in engineering such systems arise from several phenomena, one of the most perplexing being the dynamic environments in which they must function. Systems which operate in dynamic environments may have unknown worst-case scenarios, may have large variances in the sizes of the data and event sets that they process (and thus, have large variances in execution latencies and resource requirements), and cannot be

characterized (accurately) by constants, by intervals or even by time-invariant statistical distributions.

This paper presents techniques for specification and modeling of real-time Quality-of-Service for distributed systems that operate in dynamic environments. The techniques are based on a programming-language-independent meta-language for describing real-time QoS in terms of end-to-end paths through application programs. Constructs are provided for the specification and modeling of deterministic, stochastic, and dynamic characteristics of environment-dependent paths. The provision for description of periodic, transient and hybrid (transient-periodic) paths is also made. Another novel feature of the language is that it allows the description of multi-level timing constraints through (1) simple, cycle deadlines (which apply to a single cycle of a periodic, transient or transient-periodic path) and (2) super-period deadlines (which apply to multiple cycles of a transient-periodic path). The language and model also consider the scalability and fault tolerance features of the end-to-end paths and their application program constituents.

This is significantly different from other real-time application development languages and other real-time meta-languages. Real-time application development languages (e.g., Tomal [12], Pearl [17], Real-Time Euclid [13], RTC++ [9], Real-Time Concurrent C [7], Dicon [14], Chaos [3], Flex [16], TCEL [5], Ada95 [2], and MPL [19]) include a wide variety of features that allow assertion checking and code transformation to ensure adherence to timing constraints. Specification languages or “meta-languages” (such as ACSR [4], GCSR [1], [23], CaRT-Spec [27] and RTL [10]) formalize the expression of timing constraints and in some cases allow proofs of properties of programs based on these constraints. In [6], the features of [10] have been folded into an application development language. Our work is a specification meta-language that is application-language-independent. Rather than providing real-time support within a particular application language, it provides support for expressing timing

constraints for *systems of application programs*, written in a melange of programming languages. Unlike previous work, in which timing constraints are associated at a relatively small granularity (e.g., task-level), our language allows timing constraints to be expressed the granularity of end-to-end *paths* which span many communicating programs.

Another way of categorizing real-time languages and models is in the way they permit characterization of a system's behavior when interacting with the physical environment. Prior work has typically assumed that the effects of the environment on the system can be modeled deterministically. Our work expands this to include systems that interact with environments that are either deterministic, stochastic, or dynamic. This is done by modeling interactions with the environment as data and event streams, that may have deterministic, stochastic or dynamic properties.

In order to handle dynamic environments, it is useful if the language includes features that can be related to dynamic mechanisms (such as those described in [20] [8] [21] [22] [24]) for monitoring, diagnosis and recovery. Language support for run-time monitoring of real-time programs has been addressed in [20], [6], [11], [13], and [2]. However, limited support was provided for diagnosis and recovery actions. Our work extends the language features pertaining to diagnosis of timing problems, and the migration or replication of software components to handle dynamic data stream or event stream loads (scalability). The specification language also provides support for fault-tolerance, an issue that has not been addressed in most real-time languages.

Previous real-time languages allow the description of behaviors that are purely periodic or aperiodic. Our work extends language support to describe hybrid behaviors such as the transient-periodic described in [26]. We also allow for multi-dimensional timing constraints, which, to our knowledge, cannot be described in any existing real-time language.

The remainder of the paper presents the specification language and shows how it is used for adaptive QoS management. Section 2 explains the software paradigm used in the specification language. In Section 3, we present the grammar of the specification language and illustrate its use with examples. Section 4 presents a static system model that is constructed from specifications. Section 5 defines a dynamic model, which is built at run-time by augmenting the static model with environment-dependent state. Use of the dynamic and static system models for adaptive resource and QoS management is illustrated in Section 6. Section 7 presents experimental results of applying the language, the models and the resource and QoS management techniques to a distributed real-time application.

2. The dynamic real-time path paradigm

This section presents the *dynamic real-time path paradigm*. A path-based real-time subsystem (see Figure 1) typically consists of a situation assessment path, an action initiation path and an action guidance path. The paths interact with the environment via evaluating streams of data from sensors, and by causing actuators to respond (in a timely manner) to events detected during evaluation of sensor data streams. The system operates in an environment that is either deterministic, stochastic or dynamic. A deterministic environment exhibits behavior that can be characterized by a constant value (see [19]). A stochastic environment behaves in a manner that can be characterized by a statistical distribution (see [15]). A dynamic environment (e.g., a war-fighting environment) depends on conditions which cannot be known in advance (see [22], [29]).

A (partial) air defense subsystem can be modeled using three dynamic paths: *threat detection*, *engagement*, and *missile guidance*. The *threat detection* path examines radar sensor data (radar tracks) and detects potential threats. The path consists of a radar sensor, a sensor data stream, a filtering program and an evaluation program. When a threat is detected and confirmed, the *engagement* path is activated, resulting in the firing of a missile to engage the threat. After a missile is in flight, the *missile guidance* path uses sensor data to track the threat, and issues guidance commands to the missile. The *missile guidance* path involves sensor hardware, software for filtering, software for evaluating & deciding, software for acting, and actuator hardware.

The remainder of this section describes the features of dynamic real-time *paths*. Recall that a path may be one of three types: situation assessment, action initiation, or ac-

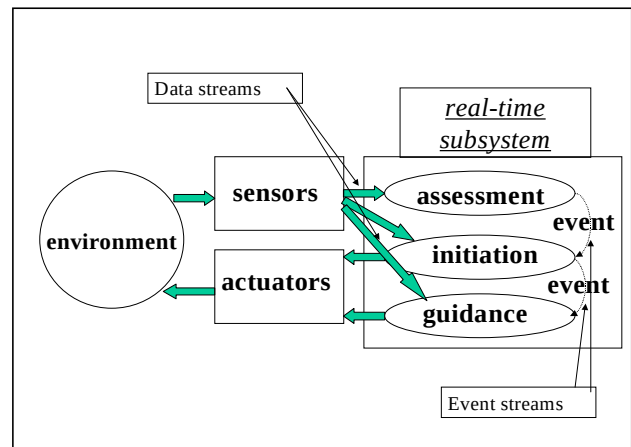


Figure 1. A real-time subsystem.

tion guidance. The first path type, situation assessment, *continuously* evaluates the elements of a sensor data stream to determine if environmental conditions are such that an action should be taken (if so, the action initiation path is notified). Thus, this type of path is called *contin-*

ous. Typically, there is a timeliness objective associated with completion of one review cycle of a continuous path, i.e., on the time to review all of the elements of one instance of a data stream. (Note: A data stream is produced by sampling the environment. One set of samples is called a data stream instance.)

The *threat detection* path of the air defense system is an example of a continuous path. It is a sensor-data-stream-driven path, with desired end-to-end cycle latencies for evaluation of radar track data. If it fails to meet the desired timeliness Quality of Service in a particular cycle, the path must continue to process track data, even though desired end-to-end latencies cannot be achieved. Peak loads cannot be known in advance for the *threat detection* path, since the maximum number of radar tracks can never be known. Furthermore, average loading of the path is a meaningless notion, since the variability in the sensor data stream size is very large (it may consist of zero tracks, or it may consist of thousands of tracks).

The second path type, action initiation, is driven by a stream of events sent by a continuous (situation assessment) path. It uses inputs from sensors to determine which actions should be taken and how the actions should be performed, notifies actuators to start performing the actions, and informs the action guidance path that an action has been initiated. We call this type of path *transient*, since it performs work in response to events. Typically, a timing objective is associated with the completion of the initiation sequence. The importance of the timing objective for a transient path is often very high, since performance of an action may be mission-critical or safety-critical.

For example, the *engagement* path of the air defense example is a transient path. It is activated by an event from the *threat detection* path, and has a QoS objective of end-to-end timeliness. The real-time QoS of this path has a higher priority than the real-time QoS of the continuous *threat detection* path.

The third path type, action guidance, is activated by an action initiation event, and is deactivated upon completion of the action. Action guidance repeatedly uses sensor inputs to monitor the progress of an actuator, to plan corrective actions needed to guide the actuator to its goal, and to issue commands to the actuator. This type of path is called *quasi-continuous*, since it behaves like a continuous path when it is active. A quasi-continuous path has *two timeliness objectives*: (1) cycle completion time: the duration of one iteration of the “monitor, plan, command” loop, and (2) action completion time (or deactivation time): the time by which the action must complete in order for success. Note that it is more critical to perform the required processing before the action completion deadline than it is to meet the completion time requirement for every cycle (although the two deadlines are certainly related). Thus, it is acceptable for the completion times of some cycles to violate the cycle deadline requirement, as long as the de-

sired actions are successfully completed by the deactivation deadline.

The *missile guidance* path of the air defense example is a quasi-continuous path. It is activated by the *missile launch* event. Once activated, the path continuously issues guidance commands to the missile until it detonates (the deactivation event). The required completion time for one iteration is dynamically determined, based on characteristics of the threat. If multiple threat engagements are active simultaneously, the threat engagement path is responsible for issuing guidance commands to all missiles that have been launched.

3. Grammar of the specification language

This section presents a specification language for describing the characteristics and requirements of dynamic, path-based real-time systems. The language provides abstractions to describe the properties of the software, such as hierarchical structure, inter-connectivity relationships, and run-time execution constraints. It also allows description of the physical structure or composition of the hardware such as LANs, hosts, interconnecting devices or ICs (such as bridges, hubs, and routers), and their statically known properties (e.g., peak capacities). Further, the Quality-of-Service requirements on various system components can be described. In this paper, we illustrate the concrete syntax to describe only the software and the Quality-of-Service objectives.

As shown in Figure 2, a subsystem is specified by describing its priority, sets of constituent applications and devices, a set of end-to-end real-time path definitions, and a graph representing the communication connectivity of the applications and devices. The non-terminals of the subsystem grammar are explained in a depth-first manner in the remainder of this section.

As seen in the grammar of Figure 2 and the example of Figure 3, the attributes of an application include boolean variables that indicate whether the application (1) can be replicated for survivability and (2) can support scalability via load sharing among replicas. Indications of whether an application combines its input stream (which may be received from different predecessor applications and/or devices), and splits its output stream (which may be distributed to different successor applications and/or devices) are also specified. Application attributes also include all information necessary to startup and shutdown applications (not elaborated in this paper). The connectivity of the subsystem describes the flow of data and events between the applications and devices of the subsystem, and is described as a sequence of ordered application and/or device pair names within parentheses. Alternately, one may use the square bracket set notation, which is a short hand to represent a complete graph.

The definition of a path (see Figure 4) includes a set of constituent applications, various attributes, QoS requirements and data/event stream definitions. The attributes of a path include priority, type, and importance. Path type, which defines the execution behavior of the path, is either continuous, transient, or quasi-continuous. The importance attribute (a string) is interpreted as the name of a dynamically linked library procedure that may be passed arguments such as priority and the current time, and returns an integer value that represents the importance of the path.

A real-time QoS specification includes timing con-

<sw-sub-system>::	Subsystem ID "{" Priority INT_LITERAL ";" <appln-defn>+ <device-defn> { <path-defn> <connectivity-descr> } * <connectivity-descr> }"
<appln-defn>::	Application ID "{" Scalable STRING ";" Survivable STRING ";" Combining STRING ";" Splitting [NONE EQUAL STRING] ";" <startup-block>+ <shutdown- block>+ }"
<device-defn>	Device { ID ";" ID ";" λ
<connectivity-descr>::	Connectivity "{" <graph-defn>+ }"
<graph-defn>::	<pair-wise-descr> <complete-graph-descr>
<pair-wise-descr>::	ID
<complete-graph-descr>::	"{" ID ";" ID ";" "} {" ID }+ }"

Figure 2. Grammar for subsystems.

Application	FilterManager { Scalable NO; Survivable YES; Combining NO; Splitting EQUAL; Startup{ ... }; Shutdown { ... } }
Device	Sensor, Actuator;
Connectivity	{ (Sensor, FilterManager) (FilterManager, Filter) (Filter, EvalDecideManager) (EvalDecideManager, EvalDecide) (EvalDecide, ActionManager) (EvalDecide, MonitorGuideManager) (ActionManager, Action) (Action, Actuator) (MonitorGuideManager, MonitorGuide) }

Figure 3. Application, device and connectivity specifications.

straints such as simple deadlines, inter-processing times, throughputs, and super-period deadlines. A simple deadline is defined as the maximum end-to-end path latency during a cycle of a continuous or quasi-continuous path, or during an activation of a transient. Inter-processing time is defined as a maximum allowable time between processing of a particular element of a continuous or quasi-continuous path's data stream in successive cycles. The throughput requirement is defined as the minimum number of data items that the path must process during a unit period of time. A super-period deadline is defined as the maximum allowed latency for all cycles of a quasi-continuous path. A super-period deadline is specified as the name of a dynamically linked library procedure that is called dynamically to determine the estimated super-period deadline. Each timing constraint specification may also include items that relate to the dynamic monitoring of the constraint. These include minimum and maximum slack values (that must be maintained at run-time), the size of a moving window of measured samples that should be ob-

<path-defn>::	Path ID "{" <apn-set> <path-attrs> <Real-TimeQoS> <SurvivabilityQoS> <stream-defns> }"
<apn-set>::	Contains "{" { <app-name> ";" } * <app-name> }"
<app-name>::	ID
<path-attrs>::	Priority INT_LITERAL ";" Type <path-type> ";" Importance STRING ";"
<path-type>::	Continuous Transient Quasi-Continuous
<Real-TimeQoS>::	Real-TimeQoS "{" <RTQoS-metric>+ }"
<RTQoS-metric>::	SimpleDeadline FLOAT_LITERAL ";" <threshold> Inter-ProcessingTime FLOAT_LITERAL ";" <threshold> Throughput INT_LITERAL ";" <threshold> Super-PeriodDeadline STRING ";" <threshold> MaxSlack INT_LITERAL ";" MinSlack INT_LITERAL ";" MonitorWindowSize INT_LITERAL ";" Violations INT_LITERAL ";" λ
<SurvivabilityQoS>::	SurvivabilityQoS "{" Survivable STRING; MinCopies INT_LITERAL ";"
<stream-defns>::	<DataStream-defn> <EventStream-defn> <DataStream-defn> <EventStream-defn>

Figure 4. Grammar for paths.

Path Sensing {	
Contains {Sensor, FilterManager, Filter, EvalDecideManager, EvalDecide}	
Type Continuous; Priority 2; Importance "CalcImport";	
Real-TimeQoS {SimpleDeadline 4; //secs	
MaxSlack 80; MinSlack 20; //PERCENT	
MonitorWindowSize 20; Violations 15;	
Inter-ProcessingTime 7; // secs	
Throughput 200; // data elements per second }	
SurvivabilityQoS { Survivable YES; MinCopies 2; }	

Figure 5. A path specification.

served, and the maximum tolerable number of violations (within the window).

Figure 4 also contains the grammar for describing survivability QoS. A survivability QoS specification includes a boolean variable that indicates (1) whether the path should be managed to ensure survivability and (2) the minimum required level of redundancy. Note that replicating a path entails replicating all of the applications that make up the path.

An example path specification is given in Figure 5. The "Sensing" path is continuous and has a priority of 2. It has a Simple Deadline of 4 seconds, which means that each review cycle must not have a latency that exceeds this amount. The MaxSlack and MinSlack definitions further constrain this requirement to the interval [0.8seconds, 3.2seconds]. If 15 out of the most recent 20 cycles violate this requirement, then corrective action by a resource manager is required. If the upper bound is exceeded, the corrective action would be to allocate more resources for the path. In the case where the lower bound is exceeded, the action would be to take away some of the resources of the path. The path inter-processing time is 7 seconds, meaning that no more than this amount of time should elapse between reviews of a particular data stream element in successive cycles. The path throughput must be at least 200 data stream elements per second. The path also requires

DataStream { Type Deterministic; Size 40;}	DataStream { Type Stochastic; Size "/home/ uta/MGDataStream.data";}
EventStream { Type Dynamic; }	EventStream { Type Stochastic; Rate "Exponential 0.5"; }

Figure 7. Specifications of streams.

<DataStream-defn>:: DataStream "[" Type [Deterministic Stochastic Dynamic] "," Size <environ-descr> "," "]"	
<EventStream-defn>:: EventStream "[" Type [Deterministic Stochastic Dynamic] "," Rate <environ-descr> "," "]"	
<environ-descr>:: INT_LITERAL "(" INT_LITERAL "," INT_LITERAL ")" <pdf-descr> λ	
<pdf-descr>:: STRING FILENAME	

Figure 6. Grammar for streams.

the existence of at least 2 copies of each of its member applications at all times.

Figure 6 shows the grammar for describing stream properties. A corresponding specification is illustrated in Figure 7. The stream type can be deterministic, stochastic, or dynamic. The data stream size or event arrival rate of a deterministic stream is a constant (scalar or interval). A stochastic stream has a data stream size or an event arrival rate that is characterized by a probability distribution function. The distribution is described as a string containing the name of a distribution and its parameters, or is described as the name of a data file containing a data set that characterizes the stream's behavior. The data stream size or event arrival rate of a dynamic stream is not described in the specification, since it must be observed at run time.

4. Static system model

This section presents an overview of a static system model (which is constructed by a specification compiler) for a single subsystem.

A software subsystem, **SS**, consists of a set of applications $\mathbf{SS.A} = \{a_1, a_2, \dots\}$, a set of devices (sensors and actuators) $\mathbf{SS.D} = \{d_1, d_2, \dots\}$, a communication graph for applications and devices $\Gamma(\mathbf{SS}) \in \Pi((\mathbf{SS.D} \cup \mathbf{SS.A}) \times (\mathbf{SS.D} \cup \mathbf{SS.A}))$, and a set of paths $\mathbf{SS.P} = \{P_1, P_2, P_3, \dots\}$. (Note: Π denotes power set).

Each path P_i is represented as a type $\tau(P_i) \in \{c, qc, t\}$ (note that 'c', 'qc' and 't' denote 'continuous', 'quasi-continuous' and 'transient', respectively), a set of applications $P_i.A = \{a_{i,1}, a_{i,2}, \dots\}$ (where $P_i.A \subseteq \mathbf{SS}(P_i).A$), a set of devices $P_i.D = \{d_{i,1}, d_{i,2}, \dots\}$ (where $P_i.D \subseteq \mathbf{SS}(P_i).D$), a communication graph $\gamma(P_i) \in \Pi((P_i.D \cup P_i.A) \times (P_i.D \cup P_i.A))$ (note that $\gamma(P_i) \subseteq \Gamma(\mathbf{SS}(P_i))$), a data stream $P_i.DS$ (defined if $\tau(P_i) \in \{c, qc\}$), and an event stream $P_i.ES$ (defined if $\tau(P_i) \in \{t, qc\}$). $\mathbf{SS}(P_i)$ denotes the subsystem in which path P_i is contained, $\mathbf{ROOT}(G(P_i))$ is used to denote the root application node of $\gamma(P_i)$ (the node which

receives a data stream from a sensor or an event stream from an application) and $\mathbf{SINK}(G(P_i))$ is used to represent the sink application of $\gamma(P_i)$ (the application which communicates with an actuator or sends an event to the root application of another path). The type of P_i 's datastream is defined as $\tau(P_i.DS) \in \{\text{dynamic, stochastic, deterministic}\}$. $\sigma(P_i.DS)$ denotes the size characteristics of P_i 's datastream.

The real-time QoS requirements of a path are: required latency of $\lambda_{\text{REQ}}(P_i)$ seconds, required throughput of $\theta_{\text{REQ}}(P_i)$ data stream elements/time, and required data inter-processing time of $\delta_{\text{REQ}}(P_i)$ (a maximum allowable time between processing of a particular element of $P_i.DS$ in successive cycles). To mask momentary QoS-spikes during QoS monitoring, a specification may define a sampling window and a maximum number of QoS violations to be tolerated within the window. $\omega(P_i)$ models the sampling window size and $\nu(P_i)$ represents the maximum allowable number of violations (within sampling window $\omega(P_i)$). $\Psi(P_i) = [\Psi_{\min}(P_i), \Psi_{\max}(P_i)]$ is the required slack interval for each QoS requirement; i.e., it is required that the ratio (required QoS - actual QoS) : (required QoS) be no less than $\Psi_{\min}(P_i)$ and no greater than $\Psi_{\max}(P_i)$.

REPLICABLE($a_{i,j}$) --- has a true or false value to indicate if application $a_{i,j}$ can be replicated for the purpose of load sharing. **COMBINING**($a_{i,j}$) --- has the value of true or false, indicating if application $a_{i,j}$ combines the inputs from its predecessors before passing the values to successor(s). **SPLITTING**($a_{i,j}$) $\in \{\text{none, equal, non-equal}\}$ indicates whether an application splits its outputs among its successors, and if so, whether the outputs equally.

5. Dynamic system model

This section presents a model of such dynamic features as the current environment, its effect on QoS, a mapping of software to computation and communication hardware, and the number of replicas of each application.

The set of replicas of $a_{i,j}$ during cycle 'c' of P_i is defined as **REPLICAS**($a_{i,j}, c$) = $\{a_{i,j,1}, a_{i,j,2}, \dots\}$. The host to which $a_{i,j,k}$ is assigned during cycle 'c' of P_i is defined as **HOST**($a_{i,j,k}, c, P_i$), and the communication path (set of LANs and interconnection devices (ICs)) used during cycle 'c' of P_i for messages between applications $a_{i,j,x}$ and $a_{i,j,y}$ is represented as **COMPPATH**($a_{i,j,x}, a_{i,j,y}, c, P_i$).

The set of elements that constitutes a data stream can vary dynamically. $P_i.DS(c) = \{P_i.DS(c)_1, P_i.DS(c)_2, \dots\}$ represents the set of elements in $P_i.DS$ during cycle 'c' of P_i . The **tactical load** (in number of data stream elements processed) of a (quasi)-continuous path P_i during its c^{th} cycle is $|P_i.DS(c)|$. The processing of elements of a data stream may be divided among replicas of an application to exploit concurrency as a means of decreasing execution latency of a path. In successive stages of a path that has

non-combining applications (applications which, after processing data received from a single predecessor, simply divide the data among their successors), data will arrive in batches to applications; hence, each application may process several batches of data during a single cycle. Thus, the model represents the batches of data processed by application/replica 'a' during cycle 'c' as $P_i.DS(c, a) = \{P_i.DS(c, a)_1, P_i.DS(c, a)_2, \dots\}$. The cardinality $|P_i.DS(c, a)|$ is called the tactical load of 'a', and can be calculated by considering the REPLICAS, SPLITTING, and COMBINING attributes of the system model. The data stream elements contained in the j^{th} batch of 'a' are denoted by $P_i.DS(c, a, j) = \{P_i.DS(c, a, j)_1, P_i.DS(c, a, j)_2, \dots\}$.

6. Dynamic QoS analysis techniques

This section illustrates the use of the dynamic and static system models for adaptive QoS management.

Monitoring of real-time QoS involves the collection of time stamped events sent from applications. The times when application/replica 'a' starts and ends processing batch 'j' of data during cycle 'c' of P_i are denoted by $s(P_i.DS(c, a, j))$ and $e(P_i.DS(c, a, j))$, respectively. The times when 'a' starts and ends processing of data stream element $P_i.DS(c, a)_k$ are represented as $s(P_i.DS(c, a)_k)$ and $e(P_i.DS(c, a)_k)$, respectively.

Observed real-time QoS metrics are defined in terms of these basic events as follows: (1) latency of path P_i during cycle 'c' = $\lambda_{OBS}(P_i, c) = \{e(P_i.DS(c, a_{i,m,n}, j)) - s(P_i.DS(c, a_{i,x,1}, 1)) \mid a_{i,m} = SINK(\gamma(P_i)), a_{i,x} = ROOT(\gamma(P_i))\}$; (2) throughput of P_i during cycle 'c' = $\theta_{OBS}(P_i, c) = |P_i.DS(c)| / \max(\lambda_{OBS}(P_i, c))$; (3) data-inter-processing time of application 'a' for datum $P_i.DS(c, a)_k$ during cycle 'c' = $\delta_{OBS}(P_i.DS(c, a)_k) = s(P_i.DS(c, a)_k) - s(P_i.DS(c-1, a)_k)$, for $c > 1$; (4) data-inter-processing time of application 'a' during cycle 'c' = $\delta_{OBS}(P_i.DS(c, a)) = f(\delta_{OBS}(P_i.DS(c, a)_k))$, for all 'k' in the range $[1, |P_i.DS(c, a)|]$. (E.g., f can be defined as 'average' or as 'max'.); (5) data-inter-processing time of P_i during cycle 'c' = $\delta_{OBS}(P_i.DS(c)) = f(\delta_{OBS}(P_i.DS(c, a)))$, for all 'a' in $P_i.A$. (E.g., f can be defined as 'average' or as 'max'.)

Analysis of a time series of the real-time QoS metrics enables detection of QoS violations. An **overload** of a path occurs in any cycle 'c' wherein $v(P_i) \leq \{d: c-d+1 < \omega(P_i) \wedge [\lambda_{REQ}(P_i) - \max(\lambda_{OBS}(P_i, c))] / \lambda_{REQ}(P_i) < \psi_{min}(P_i) \}$. Similarly, an **underload** of a path occurs in any cycle 'c' wherein $v(P_i) \leq \{d: c-d+1 < \omega(P_i) \wedge [\lambda_{REQ}(P_i) - \max(\lambda_{OBS}(P_i, c))] / \lambda_{REQ}(P_i) > \psi_{max}(P_i) \}$.

When a path-level (end-to-end) real-time QoS violation occurs, diagnosis determine the cause(s) of the violation (i.e., identifies subpaths (application programs) that are experiencing significant slowdown). One diagnosis technique declares an application/replica 'a' to be unhealthy during cycle 'c' of path P_i if there exists another cycle 'd'

such that the following conditions hold. **Condition 1:** $d < c$. **Condition 2:** $HOST(a, c, P_i) = HOST(a, d, P_i)$. **Condition 3:** $|P_i.DS(c, a)| = |P_i.DS(d, a)|$. **Condition 4:** $\forall f: (f < c) \wedge [HOST(a, c, P_i) = HOST(a, f, P_i)] \wedge [|P_i.DS(c, a)| = |P_i.DS(f, a)|] \wedge \max(\lambda_{OBS}(P_i, f)) > \max(\lambda_{OBS}(P_i, d))$. **Condition 5:** $\max(\lambda_{OBS}(P_i, d)) < \max(\lambda_{OBS}(P_i, c)) - \epsilon$. Note: ϵ is the minimal difference between cycle latencies that is considered significant.

Following diagnosis, allocation analysis identifies and ranks recovery actions for the applications/replicas diagnosed as unhealthy. The first step of allocation analysis is to determine if the tactical load of an unhealthy application has significantly increased in the recent past. The tactical load of an application/replica 'a' has increased in the α cycles immediately prior to cycle 'c' by at least a significant amount ' Δ ' when: $\exists d: c > d \wedge c-d < \alpha+1 \wedge (|P_i.DS(c, a)| > |P_i.DS(d, a)| + \Delta)$. The tactical load change analysis is used to determine the appropriate allocation action for an unhealthy application as follows. **Replicate** an unhealthy application/replica 'a' when its tactical load has increased recently by a significant amount. We denote such an action as: $act(a) = (a, replicate)$. **Move** an 'a' when its tactical has not increased recently by a significant amount. We denote such an action as: $act(a) = (a, move)$. We define $ACT = \{act_1, act_2, \dots\}$ as the set of actions selected, where act_i is a pair **(a, action)** and denotes that 'action' is recommended for 'a'.

The second step of allocation analysis is to group actions which address the same cause of poor path QoS. All actions that move applications/replicas from a particular host are grouped, since any one of those actions will reduce the contention experienced by applications on the host. All actions that involve replication of a particular application 'a' are also grouped, since the addition of another replica of 'a' will cause a redistribution of load processed by the existing replicas of 'a'.

In addition to grouping related actions, the unhealthy applications are ranked in decreasing order by the degree of slowdown they are experiencing. The slowdown experienced by application 'a' during cycle 'c' of P_i is computed as $\xi(a, c, P_i) = \max(\lambda_{OBS}(P_i, c)) / \max(\lambda_{OBS}(P_i, d))$, where 'd' is the cycle wherein 'a' exhibited its 'best' performance over all cycles wherein 'a' had the same tactical load as in cycle 'c'.

7. Experimental Results

This section presents two scenarios showing system behavior for the 'benchmark' situation assessment path (see Section 2, and also Figure 5) under different run-time scenarios. The path is hereafter referred to as the detect/sensing path. In the run-time scenarios, the adaptive resource management middleware monitors the real-time

QoS of the benchmark application, detects QoS violations, diagnoses cause(s) of poor QoS, and recovers accordingly.

Figure 8 depicts the overall architecture of the adaptive resource management system. In its current implementation, resource management is activated in 3 modes: during the initial system start-up process (to start application programs), when a path becomes unhealthy (i.e., a path latency exceeds the required deadline), and when an application program is terminated (due to hardware/software faults). A description of each of these resource management modes follows.

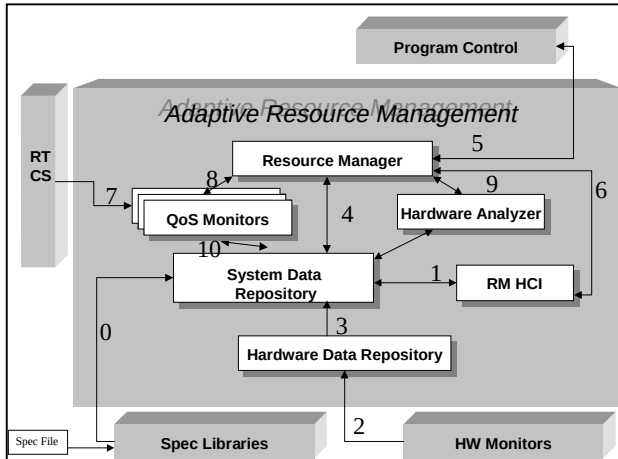


Figure 8. The resource management architecture.

The actions performed in *start-up mode* are as follows:

- (1) The System Data Repository (SDR) loads the user system specification file (via a compiler) into data structures (step 0 of Figure 8).
- (2) The SDR sends the compiled system description to the Resource Management (RM) HCI for display (step 1).
- (3) Hardware (HW) Monitors (which run on each host) periodically calculate load indices for hosts and LANS, and pass this information to the Hardware Data Repository (HDR) (step 2), which in turn passes such information to the SDR (step 3).
- (4) RM receives the system startup information from the SDR (step 4).
- (5) RM determines the initial hardware-to-software allocation and informs Program Control (PC) to start the Real-Time Control System (RTCS) application programs on specific hosts and LANs (step 5).
- (6) PC starts the programs (via daemons running on each host) and informs RM when finished (step 5).
- (7) RM sends startup information to the RM HCI for display (step 6).
- (8) RTCS application programs periodically send application profile information (time stamps and/or program/path latencies) to QoS Monitors (step 7). Global time is provided to the RTCS and to the QoS monitors via Network Time Protocol (NTP) [18].

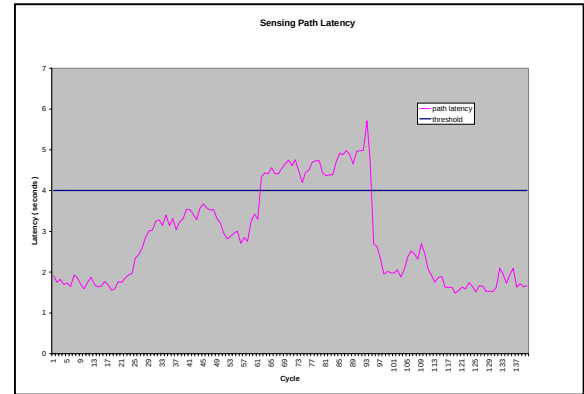


Figure 9. Sensing path latency (scenario I).

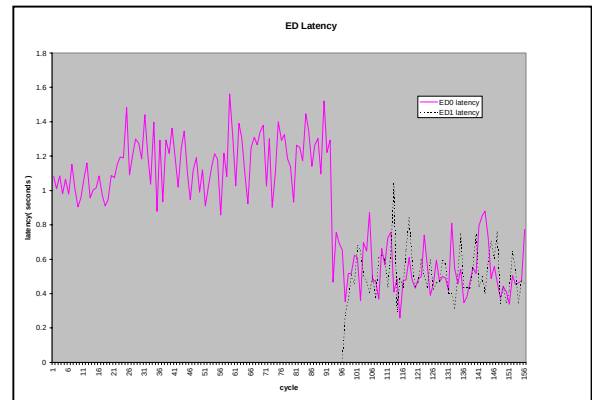


Figure 10. Latencies of the *filter* subpaths.

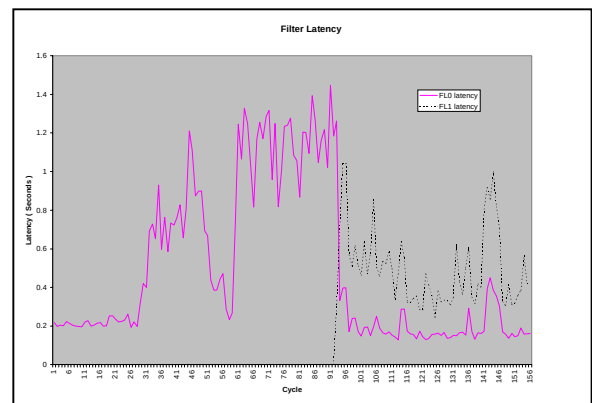


Figure 11. Latencies of *evaluate* subpaths.

The sequence performed during *QoS monitoring and enforcement mode* is: (1) If a path becomes unhealthy (misses its deadline), the QoS Monitors detect such a condition, diagnose the cause of poor health, and suggest actions (such as moving and/or replicating an application program(s)) to RM (step 8 of Figure 8). (2) RM selects the host onto which the unhealthy sub-path(s) needs to be replicated or moved, by choosing the host with the smallest load index. To enable this, the Hardware Analyzer (HA) ranks the hosts in ascending order by load index [25] and passes this information to the Resource Manager (step 9). (3) RM requests PC to enact the new allocation (step 5) and updates the RM HCI accordingly (step 6).

The actions taken in *program recovery mode* are: (1) If a sub-path (RTCS program) is terminated due to a hardware/software failure, PC detects this and informs RM (step 5 of Figure 8). (2) RM finds a healthy host with the smallest load index by querying HA (step 9). (3) RM restarts the terminated program on the healthy host by informing PC (step 5), and also notifies RM HCI to update the display (step 6 of Figure 8). To avoid thrashing via restarting a faulty piece of software ad infinitum, this step is only repeated a fixed number of times.

The remainder of this section describes two scenarios depicting the system behavior under different run-time conditions. The experiments were performed on a distributed network consisting of two Sun Sparc5 workstations (170MHZ processor, 32 MB RAM, 2GB disk, and Sun Solaris 2.5.1 configurations), named virginia and nujersy; and two Sun Ultra Sparc I workstations (167MHZ processor, 128MB RAM, 2GB disk, and Sun Solaris 2.5.1 configurations) named texas and desidrta. The hosts are connected via a 10/100MB per second ethernet network. The RTCS application consists of the Sensing path of the benchmark application with a simple deadline, $\lambda_{REQ}(Sensing)$, of 4 seconds. The sensing path consists of a set of programs (called computation subpaths) that may be distributed, and which cooperate in pipeline fashion. The programs are: Sensor (a simulated radar), Filter Manager (FM), Evaluate and Decide Manger (EDM), Filter (FL), and Evaluate and Decide (ED).

The first experiment scenario is depicted in Figure 9, which shows a plot of the observed Sensing path latency ($\lambda_{OBS}(Sensing, c)$) and shows $\lambda_{REQ}(Sensing)$ as a straight line. Initially, Sensor, FM, and EDM run on virginia, one copy of FL runs on texas, and one copy of ED runs on nujersy. RM and a QoS Monitor run on nujersy, SDR and PC run on texas, and HDR and HA run on desidrta.

As shown in Figure 9, the data stream size (i.e., the tactical load) of the sensing path during the initial execution cycles $c \in [1, 23]$ is 2000 elements; that is $|Sensing.DS(c)| = 2000$. The path latencies during this interval are below the required QoS threshold, i.e., $\lambda_{OBS}(Sensing, c) < \lambda_{REQ}(Sensing)$, for $c \in [1, 23]$. At cycle $c=23$ the tactical load is incremented by 500, and the path latency increases.

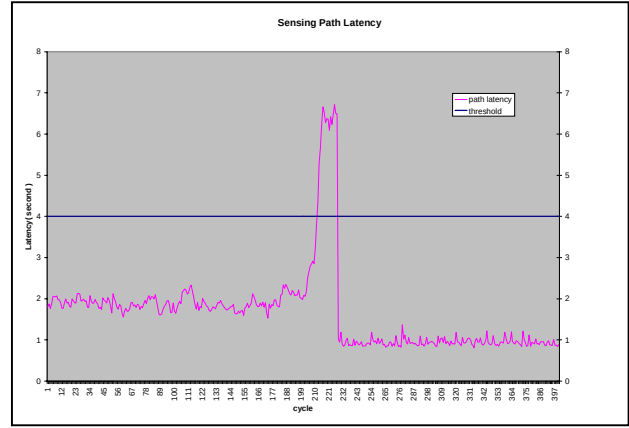


Figure 12. Sensing path latency (Scenario II).

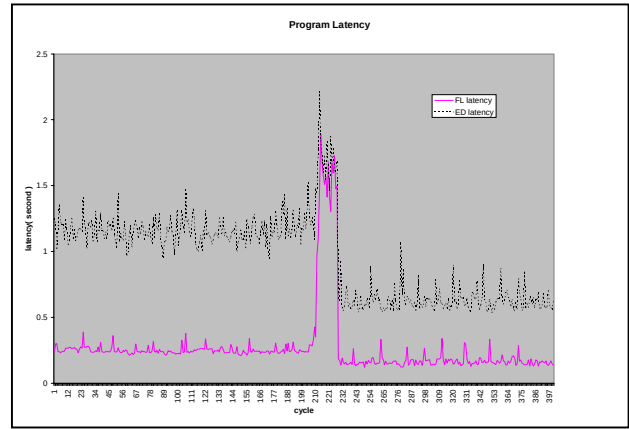


Figure 13. Latencies of FL and ED subpaths.

The tactical load is also incremented by 500 at cycle 55. From this cycle onward, the path latency gradually increases and finally exceeds the QoS threshold. Figures 10 and 11 show the corresponding latencies of the FL and ED subpaths, respectively. Increases in the latencies of these programs are seen to correspond with the increases in the tactical load of the path.

The QoS Monitor component of the adaptive resource management system detects the QoS violation during the [59, 79] cycle interval (shortly after the QoS violation as shown in Figure 9). In both of these experiments the sampling window size is 20 ($\omega(Sensing)=20$), the number of acceptable violations per window is 15 ($\nu(Sensing)=15$), and the minimum slack is 0% ($\psi_{min}(Sensing)=0\%$). The QoS Monitor diagnoses the cause of poor health in cycle 79 and recommends replications of the FL and ED programs. The path monitor function has a latency of 1.135 μ sec. and the path diagnosis function has a latency of 1.86 μ sec. RM activates a copy of ED (ED1) on nujersy and a copy of FL (FL1) on texas at cycle 79. The latencies of

RM and PC are 211.2 and 1.84 $\mu\text{sec.}$, respectively. Thus, all the resource re-allocation and enactment processes take place in one cycle of the sensing path. However, in the RTCS system, a newly activated application program takes several cycles to establish its communication sockets and thus the latency of the path does not immediately decrease. Figure 10 depicts how the latency of each copy of FL is reduced when a second copy of FL is initialized at cycle 92. Similarly, Figure 11 depicts how the latency of each copy of ED is reduced when a second copy of ED is initialized at cycle 95.

After the initialization of the replicas of the FL and ED programs, the path latency falls below the threshold around cycle 95 (Figure 9). Note that despite increasing the tactical load by a factor of 50%, the adaptive resource management system was able to adapt to this dynamic change and preserve the QoS requirement of the RTCS application.

Experiment scenario II begins with the sensing path complying with its QoS requirements. Instead of encountering a dynamic increase in tactical load, a dynamic system load increase is caused on one of the hosts, thereby increasing resource contention. The RTCS application programs are initially distributed as follows: Sensor and FL are assigned to virginia and FM, EDM, and ED are assigned to nujersy. Throughout this experiment, $|\text{Sensing.DS}(c)|=2000$, $\lambda_{\text{REQ}}(\text{Sensing})=4 \text{ sec.}$, $\alpha(\text{Sensing})=20$, $v(\text{Sensing})=15$, and $\psi_{\text{min}}(\text{Sensing})=0\%$.

As shown in Figure 12, $\lambda_{\text{OBS}}(\text{Sensing}, c) < \lambda_{\text{REQ}}(\text{Sensing})$, for $c \in [1, 208)$. At cycle 208, nujersy's load is increased by activating several computationally intensive applications. This causes $\lambda_{\text{OBS}}(\text{Sensing}, c)$ to increase and to eventually exceed the threshold of $\lambda_{\text{REQ}}(\text{Sensing})$. Correspondingly, significant increases are observed in the latencies of the FL and ED subpaths (see Figure 13).

The QoS Monitor component of the adaptive resource management system detects the QoS violation during the [208, 228] cycle interval, diagnoses the cause of poor health (also during cycle 228) and recommends moving the FM, EDM, ED, and FL programs to another host. RM consults HA for host load indexes and decides to move the FM, EDM, ED, and FL programs to host texas. Figure 13 depicts how the latencies of the FL and ED programs are reduced after the move (cycle 228). Note that since no new application is activated in this scenario, the effect of resource re-allocation is almost immediate.

Finally, after some of the RTCS programs are moved to a host with a lower load, $\lambda_{\text{OBS}}(\text{Sensing}, c)$ falls below $\lambda_{\text{REQ}}(\text{Sensing})$, for $c \geq 230$. Note that despite drastically increasing a host's load, the adaptive resource management system was able to adapt to this dynamic change and to preserve the QoS requirement of the RTCS application. It was also able to successfully diagnose causes of QoS violations.

Through the above two experiments the functionality of the resource management middleware has been validated, and its responsiveness to dynamic workload changes has been demonstrated. It should be noted that the observed variances in application latency may be improved further by the use of a real-time operating system. Additionally, it is planned to reduce the delayed effect of a reallocation action (replication) by redesigning the inter-process communication portion of the RTCS application. We have conducted several other experiments under other execution scenarios, such as simultaneous increases in tactical load and host loads, causing both moving and replicating of the programs. In every experiment, the middleware behaved as expected, successfully restoring the health of the RTCS applications (see [28] for more information).

8. Conclusions and future work

This paper has described the dynamic class of real-time systems, presented engineering techniques and tools appropriate for such systems, and demonstrated the utility of those tools for a dynamic real-time application. An important component of this paper is a specification language for real-time QoS. The language allows the description of environment-dependent application features, thereby facilitating the modeling and dynamic resource management of such systems. A model for static and dynamic aspects of such systems was presented, and dynamic QoS management techniques were defined in terms of the model. An example of the use of the specification language and the model for adaptive resource management of a dynamic real-time application was shown.

Ongoing work includes application of the tools and techniques to shipboard computing systems. Work is also being performed to develop and experimentally compare allocation algorithms and QoS forecasting algorithms.

9. Acknowledgements

This work was sponsored in part by DARPA/NCCOSC contract N66001-97-C-8250, and by NSWC/NCEE contracts NCEE/A303/41E-96 and NCEE/A303/50A-98.

The authors would like to thank scientists from NSWC for collaborative efforts which seeded the ideas presented in this paper. We are also indebted to the members of the Laboratory for Parallel and Distributed Real-time Systems at the University of Texas at Arlington.

10. References

- [1] H. B-Abdallah, I. Lee and J-Y. Choi, A Graphical Language with Formal Semantics for the Specification and Analysis of Real-Time Systems, Proceedings of The IEEE Real-Time Systems Symposium, pages 276--286, December 1995.

- [2] International Standard ANSI/ISO/IEC-8652:1995, Ada 95 Reference Manual, Intermetrics, Inc., January 1995.
- [3] T. Bihari and P. Gopinath, Object-oriented real-time systems, *IEEE Computer*, 25(12):25--32, December 1992.
- [4] J-Y. Choi, I. Lee, H-L. Xie, The Specification and Schedulability Analysis of Real-Time Systems using ACSR, *Proceedings of The IEEE Real-Time Systems Symposium*, pages 266--275, December 1995.
- [5] R. Gerber and S. Hong, Semantics-based compiler transformations for enhanced schedulability, *Proceedings of The IEEE Real-Time Systems Symposium*, pages 232--242, December 1993.
- [6] M. Gergeleit, J. Kaiser, and H. Streich, Checking timing constraints in distributed object-oriented programs, *Proceedings of The Object-Oriented Real-Time Systems (OORTS) Workshop*, October 1995.
- [7] N. Gehani and K. Ramamritham, Real-Time Concurrent C: A language for programming dynamic real-time systems, *Journal of Real-Time Systems*, 3(4):377--405, December 1991.
- [8] D. Hull, A. Shankar, K. Nahrstedt and J. W. S. Liu, An End-to-End QoS Model and Management Architecture, *Proceedings of The IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, pages 82-89, 1997. The 18th IEEE Real-Time Systems Symposium.
- [9] Y. Ishikawa, H. Tokuda, and C. M. Mercer, An object-oriented real-time programming language, *IEEE Computer*, 25(10):66--73, October 1992.
- [10] F. Jahanian and A. K.-L. Mok, Safety analysis of timing properties in real-time systems, *IEEE Transactions on Software Engineering*, 12(9):890--904, 1986.
- [11] K. B. Kenny and K. J. Lin, Building flexible real-time systems using the Flex language, *IEEE Computer*, pages 70--78, May 1991.
- [12] R. B. Kieburtz and J. L. Hennessy, Tomal - a high-level programming language for microprocessor process control applications, *ACM SIGPLAN Notices*, pages 127--134, April 1976.
- [13] E. Kligerman and A. D. Stoyenko, Real-Time Euclid: A language for reliable real-time systems, *IEEE Transactions on Software Engineering*, 12(9):941--949, September 1986.
- [14] I. Lee and V. Gehlot, Language constructs for distributed real-time systems, *Proceedings of The IEEE Real-Time Systems Symposium*, December 1985.
- [15] J. P. Lehoczky, Real-Time Queueing Network Theory, *Proceedings of The IEEE Real-Time Systems Symposium*, pages 58--67, December 1997.
- [16] K. J. Lin and S. Natarajan, Expressing and maintaining timing constraints in Flex, *Proceedings of The 9th IEEE Real-Time Systems Symposium*, pages 96--105, December 1988.
- [17] T. Martin, Real-time programming language pearl - concept and characteristics, *Proceedings of The IEEE Computer Society Second International Computer Software and Applications Conference (COMPSAC)*, pages 301--306, 1978.
- [18] D. L. Mills, Improved Algorithms for Synchronizing Computer Network Clocks, *IEEE/ACM Transactions on Networks*, 3, June 1995.
- [19] V. M. Nirkhe, S. K. Tripathi, A. K. Agrawala, Language Support for the Maruti Real-Time System, *Proceedings of The 11th IEEE Real-Time Systems Symposium*, pages 257--266, 1990.
- [20] R. Rajkumar F. Jahanian and S. Raju, Run-time monitoring of timing constraints in distributed real-time systems, *Journal of Real-Time Systems*, 1994.
- [21] R. Rajkumar, C. Lee, J. Lehoczky and D. Siewiorek, A Resource Allocation Model for QoS Management, *Proceedings of The 18th IEEE Real-Time Systems Symposium*, pages 298--307, 1997.
- [22] D. Rosu, K. Schwan, S. Yalamanchili and R. Jha, On Adaptive Resource Allocation for Complex Real-Time Applications, *Proceedings of The 18th IEEE Real-Time Systems Symposium*, pages 320--329, December 1997.
- [23] A. Shaw, Reasoning about time in higher-level language software, *IEEE Transactions on Software Engineering*, 15(7):875--889, July 1989.
- [24] K. G. Shin and C.-J. Hou, Design and Evaluation of Effective Load Sharing in Distributed Real-Time Systems, *IEEE Transactions on Parallel and Distributed Systems*, 5(7), pages 704-719, July 1994.
- [25] J. Song, Characterization of Load Indices for Unix Workstations, Master's Project, The University of Texas at Arlington, May 1998.
- [26] S. Sommer and J. Potter, Operating system extensions for dynamic real-time applications, *Proceedings of The IEEE Real-Time Systems Symposium*, pages 45--50, December 1996.
- [27] L. R. Welch, A. D. Stoyenko, and T. J. Marlowe, Response Time Prediction for Distributed Periodic Processes Specified in CaRT-Spec, *Control Engineering Practice*, 3(5), May 1995, pp. 651-664.
- [28] L. R. Welch, P. Shirolkar, B. Shirazi, et al., Adaptive Resource Management For Scalable Dependable Real-Time Systems: Middleware Services and Applications to shipboard computing systems, Technical Report, TR-CSE-97-009, The University of Texas at Arlington, December 1997.
- [29] L. R. Welch, B. A. Shirazi, B. Ravindran and C. Bruggeman, DeSiDeRaTa: QoS Management Technology for Dynamic, Scalable, Dependable, Real-Time Systems, *Proceedings of The 15th IFAC Workshop on Distributed Computer Control Systems*, September 1998.