Distributed, Scalable, Dependable Real-Time Systems: Middleware Services and Applications

Lonnie R. Welch[1], Binoy Ravindran[2], Paul V. Werme[3], Michael W. Masters[3], Behrooz A. Shirazi[1], Prashant A. Shirolkar[1], Robert D. Harrison[3], Wayne Mills[3], Tuy Do[3], Judy Lafratta[3], Shafqat M. Anwar[1], Steve Sharp[4], Terry Sergeant[5], George Bilowus[4], Mark Swick[4], Jim Hoppel[4], Joe Caruso[4]

Abstract

Some classes of real-time systems function in environments which cannot be modeled with static approaches. In such environments, the arrival rates of events which drive transient computations may be unknown. Also, the periodic computations may be required to process varying numbers of data elements per period, but the number of data elements to be processed in an arbitrary period cannot be known at the time of system engineering, nor can an upper bound be determined for the number of data items; thus, a worst case execution time cannot be obtained for such periodics. This paper presents middleware services that support such dynamic real-time systems *through adaptive resource management. The middleware services have been implemented and employed for components of the experimental Navy system described in [10]. Experimental characterizations show that the services provide timely responses, that they have a low degree of intrusiveness on hardware resources, and that they are scalable.*

1. Introduction

Many real-time systems have rigorous, multi-dimensional Quality-of-Service (QoS) objectives. They must behave in a dependable manner, respond to threats in a timely fashion and provide continuous availability within hostile 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, even though the worst case scenarios of the environment may be unknown (e.g., see [6]). This paper describes innovative QoS and resource management technology for such systems.

Our approach is based on the dynamic path paradigm. A path-based real-time subsystem (see [11]) typically consists of a detection & 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.

Most previous work in distributed real-time systems has focused on a lower level of abstraction than the path and has assumed that all system behavior follows a statically known pattern [8, 9]. When applying the previous work to some applications (such as those described in [WRHM97]), problems arise with respect to scalability of the analysis and modeling techniques; furthermore, it is sometimes impossible to obtain some of the parameters required by the models. The work described in this paper addresses these problems.

A major difference between the traditional load balancing techniques [7] and dynamic QoS-based resource management services lies in the overall goals. While load balancing systems (see Load-Leveler [5], LSF [12], NQE [2], PBS [4], Globus [3], and Condor [1]) attempt to achieve *system* performance goals such as minimized response time or maximized throughput, dynamic QoS-based resource managers strive to meet the QoS requirements of each application they manage. Another major difference between these systems is their workload models. Traditional load balancing systems assume *independent* jobs with *known resource requirements*. In a dynamic resource

[1] Computer Science & Engineering Dept.; University of Texas at Arlington; Box 19015, Arlington, TX 76019; {welch|shirazi}@cse.uta.edu
[2]The Bradley Dept. of Electrical and Computer Engineering;Virginia Polytechnic Institute and State University; Blacksburg, VA 24061;binoy@vt.edu
[3]Code B35; Naval Surface Warfare Center,Dahlgren, VA 22448; {WermePV|MastersMW}@nswc.navy.mil
[4]Computer Sciences Corporation, Dahlgren, VA 22448
[5]Ouachita Baptist University, 410 Ouachita Street, Arkadelphia, AR 71998-0001; sergeantt@alpha.obu.edu

management system, the workload requirements of applications can vary, based on environmental conditions; additionally, applications are dependent (communicate with each other).

The rest of the paper is organized as follows. Section 2 provides an overview of a middleware architecture for dynamic QoS management of path-based systems, and describes the adaptive resource allocation approach employed by the middleware. In section 3 we present our experiences with the QoS management middleware services. This includes a description of the Navy testbed in which the techniques were employed, and experimental results characterizing response times of the middleware services.

2. Dynamic resource and QoS management

The logical architecture of the QoS management software is shown in Figure 1. It behaves as follows. The application programs of real-time control paths send time-stamped events to the *metrics calculation* component, which calculates path-level QoS metrics and sends them to the *QoS monitor*. The monitor checks for conformance of observed QoS to required QoS, and notifies the *QoS diagnosis* component when a QoS violation occurs. QoS diagnosis notifies the *action selection* component of the cause(s) of poor QoS and recommends actions (e.g., move a program to a different host or LAN, shed a program, or replicate a program) to improve QoS. Action selection ranks the recommended actions, identifies redundant actions, and forwards the results to the *allocation analysis* component; this component consults resource discovery for host and LAN load index metrics, determines a good way to allocate the hardware resources in order to perform the actions, and requests the actions be performed by the *allocation enactment* component.

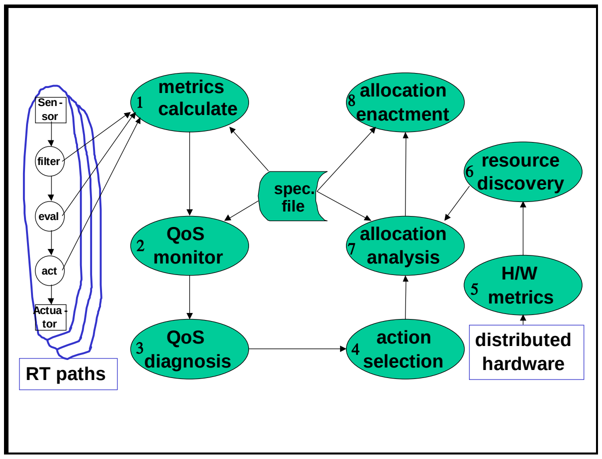

Figure 1. Logical architecture of the resource and QoS management software.

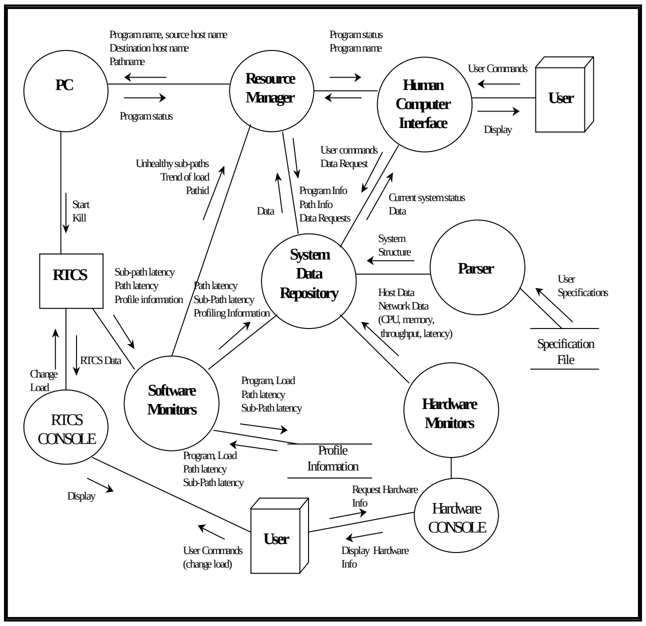

Figure 2. Physical architecture of the resource and QoS management software.

The physical QoS management architecture is shown in Figure 2. The core component of the middleware is the resource manager. It is activated when programs die and when time-constrained control paths miss their deadlines. In response to these events, it takes appropriate measures to improve the quality of service delivered to the applications. The reallocations made by the resource manager make use of information provided by the hardware and software monitors, as well as from a specification file that describes QoS requirements and the structures of the software system and the hardware system. The system data repository component is responsible for collecting and maintaining all system information. The program control (PC) component consists of a central control program and a set of startup daemons. When resource manager needs to start a program on a particular host, it informs the control program, which notifies the startup daemon on that host. The HCI provides information to the user regarding the system configuration, application status, and reallocation decisions. It also allows the operator to dynamically modify the behavioral characteristics of the resource discovery and the resource manager components.

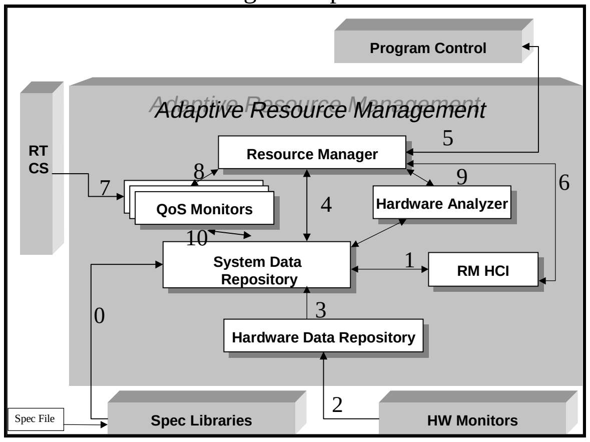

Figure 3. Adaptive resource management scenarios.

Figure 3 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.

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

1. The System Data Repository loads the user spec file via the Spec Libraries, which consist of a compiler and data structures to store the compiled real-time system and application QoS specifications (step 0 of Figure 3).
2. The System Data Repository sends the system information to Resource Management Human Computer Interface (RM HCI) for display purposes (step 1 of Figure 3).
3. Hardware (HW) Monitors continuously observe a resource's load index and pass this information to the Hardware Data Repository (step 2 of Figure 3), which in turn passes such information to the System Data Repository (step 3 of Figure 3).
4. The Resource Manager receives the initial startup information from the System Data Repository, as specified by the spec file (step 4 of Figure 3).
5. The Resource Manager informs the Program Control to start the Real-Time Control System (RTCS) application programs on the specified hosts (step 5 of Figure 3).
6. The Program Control starts the programs and informs the Resource Manger accordingly (step 5 of Figure 3).
7. The Resource Manger sends startup information to RM HCI for display purposes (step 6 of Figure 3).
8. The RTCS continuously sends the application profile (time stamps or program/path latencies) information to QoS Monitors (step 7 of Figure 3). Global time is made available to RTCS via the Network Time Protocol (NTP) package.

The sequence performed during *QoS monitoring and enforcement mode* is:

6. If a path becomes unhealthy (misses its deadline), the QoS Monitors detect such a condition, diagnose the cause of poor health, and suggest an action (such as moving or replicating an application program) to the Resource Manager (step 8 of Figure 3).
7. The Resource Manger needs to decide on which host(s) and LAN(s) the unhealthy sub-path needs to be replicated or moved. This decision is made by choosing the host(s) and LAN(s) with the smallest load indices. The Hardware Analyzer ranks the hosts and LANs in ascending order of their load index and passes this information to the Resource Manager (step 9 of Figure 3).
8. Once a host is selected, the Resource Manger notifies the Program Control to make the change (step 5 of Figure 3) and updates the RM HCI accordingly (step 6 of Figure 3).

The actions taken in *program recovery mode* are described below:

1. If a sub-path (RTCS program) is terminated due to some hardware/software failure, Program Control detects such a condition and informs the Resource Manager accordingly (step 5 of Figure 3).
2. The Resource Manger finds the host(s) and LAN(s) with the smallest load indices by querying the Host analyzer (step 9 of Figure 3).
3. It then re-starts the terminated program on that host by informing the Program Control (step 5 of Figure 3) and RM HCI (step 6 of Figure 3) accordingly. In order to avoid thrashing (restarting a faulty piece of software), this step is only repeated an operator-determined fixed number of times.

3. Experimental results

The technology described in this paper was evaluated within the Naval Surface Warfare Center High Performance Computing (NSWC HiPer-D) Testbed, which contains the experimental Navy system described in [10]. The implementation includes the following capabilities: (1) a simulated track source, (2) track correlation and filtering algorithms, (3) track data distribution services, (4) a doctrine server and three types of doctrine processing, (5) an engagement server, (6) a display subsystem including X-windows based tactical displays, submode mediation, and alert routing surface operations, (7) a simulated weapons control system, and (8) identification upgrade capabilities.

The software runs on a heterogeneous network configuration that includes Myrinet, ATM, FDDI, and ethernet, on multiple heterogeneous host platforms, including Dec Alpha's with OSF-1, a Dec Sable with OSF-1, TAC-4's with HP-UX, Sun Sparc 10's with Solaris, and Pentiums with OSF-1RT.

We performed experiments to determine the responsiveness of our QoS and resource management middleware for survivability and scalability services.

The *total Survivability response time(T_u)* calculation is divided into four major phases : (1) *Program Death Detection time (t_1)* is the time taken by the Startup Daemon to inform the Program Manager of a dead program, (2) *Resource Manager Notification time (t_2)* is the time taken by the Program Manager to inform the Resource Manager of the dead program, (3) *Resource Manager Processing time (t_3)* is the time taken by the Resource Manager to select a good host, and (4*) Restart time (T_r)* is the time taken by the Program Manager and Startup Daemons to actually restart the program.

The processing time at the resource manager, t_3 , is further decomposed. Preprocessing time (t_{31}) is the time interval after receipt of a *dead program* message from the program manager and before network discovery begins. This time interval is internal to the resource manager. Network Resource Discovery (t_{32}) is the time interval required to obtain network-level metrics from the network controller. Host Resource Discovery (t_{33}) is the time interval required to obtain host-level metrics from all eligible host monitors. Allocation Decision time (t_{34}) is the time interval required to choose a good host. This interval is internal to the resource manager. Post-processing time (t_{35}) is the time interval after finding the best host and before sending a program restart instruction to the program manager. This interval is internal to the resource manager.

The Restart time, T_r, consists of three phases. Program Notification time (t_2') is the time required to inform the Program Manager to restart a particular program on a particular host. Program Manager to Startup daemon data transfer time (t_4) is the time required to transfer the restart data to the appropriate Startup daemon from the Program Manager. Program Start Detection time (t_1') is the time required by the startup daemon to detect the start of the program.

We repeatedly measured the response times of the survivability services, and observed that the total average response time, T_u, is 3.45 seconds, with a standard deviation of 0.021435. The data also indicate that the maximum time is spent during host resource discovery, followed by the time taken by the startup daemons to actually start the program.

Response time measurements were also made for path overload detection and overload recovery via automatic scalability. The *total Scalability response time (T_c)* calculations are divided into 4 phases: (1) *Path Overload data transfer time (t_2)* is the time taken for the overloaded path information to reach the resource manager. This interval occurs immediately after the overload detection heuristic in the subsystem manager detects the overloaded condition of the path (t_1), (2) *Resource Manager processing (t_3)* is the same as in T_u, (3) *Scale time (T_s)* is the time taken by the Program Manager and Startup Daemons to actually start a new copy of the program. The scale time (T_s) consists of three phases, identical to the restart component of T_u. In repeated experiments, the average total response time, T_c, for scalability services was 4.51 seconds, with a standard deviation of 0.020731.

4. Conclusions and future work

This paper describes adaptive resource management middleware that provides integrated services for fault tolerance, distributed computing, and real-time computing. The underlying system model differs significantly from that used in related work. Furthermore, the services have been applied to an experimental Navy system prototype. Experiments show that the services provide bounded response times, scalable services, and low intrusiveness.

5. 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.

6. References

[1] "Condor Project," http://www.cs.wisc.edu/condor/, 1999.

[2] Cray Research, Document in-2153 2/97, Technical report, Cray Research, 1997.

[3] I. Foster and C. Kesselman. "Globus Project," http://www.globus.org/, 1999.

[4] R. Henderson and D. Tweten. "Portable Batch Systems: External Reference Specification," Technical report, NASA, Ames Research Center, 1996.

[5] IBM. Corporation. "*IBM Load Leveler: User's Guide*," Sept. 1993.

[6] Gary Koob, "Quorum," Proceedings of the DARPA ITO General PI Meeting, pages A-59 to A-87, October 1996.

[7] B. Shirazi, A.R. Hurson, and K. Kavi, "Scheduling and Load Balancing in Parallel and Distributed Systems," IEEE Press, 1995.

[8] S. Son, "Advances in Real-Time Systems," Prentice Hall, 1995.

[9] J. Stankovic, and K. Ramamritham, "Advances in Real-Time Systems," IEEE Computer Society Press, April 1992.

[10] L. R. Welch, B. Ravindran, R. Harrison, L. Madden, M. Masters and W. Mills, "Challenges in Engineering Distributed Shipboard Control Systems," *The IEEE Real-Time Systems Symposium*, December 1996.

[11] L. R. Welch, B. Ravindran, B. Shirazi and C. Bruggeman, "Specification and analysis of dynamic, distributed real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 72-81, IEEE Computer Society Press, 1998.

[12] S. Zhou, "LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems," *Proc. Workshop on Cluster Computing*, 1992.