A Distributed System Reference Architecture for Adaptive QoS and Resource Management

Lonnie R. Welch[1], Michael W. Masters[2], Leslie A. Madden[2], David T. Marlow[2], Philip M. Irey IV[2], Paul V. Werme[2], and Behrooz A. Shirazi[1]

Abstract. This paper deals with large, distributed real-time systems that have execution times and resource utilizations which cannot be characterized *a priori*. (The motivation for our work is provided in part by the characteristics of *combat systems*.) There are several implications of these characteristics: 1) demand space workload characterizations may need to be determined *a posteriori*, and 2) an adaptive approach to resource allocation may be necessary to accommodate dynamic workload changes. Thus, we present a scheme for dynamically managing distributed computing resources by continuously computing and assessing QoS and resource utilization metrics that are determined *a posteriori*. Specifically, the paper presents an adaptive distributed system reference architecture that is suitable for such an approach. This reference architecture provides the capabilities and infrastructure needed to construct multi-component, replicated, distributed object real-time systems that negotiate for a given level of QoS from the underlying distributed computing resources.

Introduction

The majority of real-time computing research has focused on the scheduling and analysis of real-time systems whose timing properties and execution behavior are known *a priori*. This is not without justification, since static approaches to the engineering of real-time systems have utility in many application domains [Sha91]. Furthermore, the pre-deployment guarantee afforded by such approaches is highly desirable. However, there are numerous applications which must operate in highly dynamic environments (such as battle environments), thereby precluding *accurate* characterization of the applications' properties by static models. In such contexts, temporal and execution characteristics can only be known *a posteriori.* (We assume the definition of *a posteriori* contained in [Web]: "from particular instances to a generalization; based on observation or experience; empirical; opposed to *a priori*." For contrast, we also present the definition for *a priori* from [Web]: "from a

[1] Computer Science and Engineering Dept., The University of Texas at Arlington, Arlington, TX 76019-0015, {welch|shirazi}@cse.uta.edu. Sponsored in part by DARPA/NCCOSC contract N66001-97-C-8250, and by the NSWC/NCEE contracts NCEE/A303/41E-96 and NCEE/A303/50A-98.

[2] The Naval Surface Warfare Center, Dahlgren, VA, 22448, {MastersMW|MaddenLA|WermePV|MarlowDT|IreyPM}@nswc.navy.mil.

generalization to particular instances; based on theory instead of experience or experiment." We distinguish three classes of analysis: *a priori, a posteriori* and *post mortem*. *A priori* analysis is performed before run-time, *a posteriori* analysis is performed at run-time and *post mortem* analysis is performed after run-time.) Thus, guarantees of real-time performance based on *a priori* characterizations are extraneous. However, the potential benefits of *a posteriori* approaches are significant. These benefits include the ability to function correctly in dynamic environments through adaptability to unforeseen conditions and higher *actual* utilization of computing resources.

This paper deals with large, distributed real-time systems that have execution times and resource utilizations which cannot be characterized *a priori*. (The motivation for our work is provided in part by the characteristics of *combat systems* [Har94, Wel96].) There are several implications of these requirements: 1) demand space workload characterizations may need to be determined *a posteriori* and 2) an adaptive approach to resource allocation may be necessary to accommodate dynamic workload changes. In existing real-time computing models, the execution time of a "job" is often used to characterize workload, and is usually considered to be known *a priori.* Typically, execution time is assumed to be an integer "worst-case" execution time (WCET), as in [Liu73, Wel95, Sha91]. While [Sha91] establishes the utility of WCET-based approaches by listing some of its domains of successful application, others [Leh96, Jah95, Hab90, Kuo97, Sun96, Tia95, Str97, Ste97, Liu91, Abe98, Atl98, Bra98] cite the drawbacks, and in some cases the inapplicability, of the approaches in certain domains. In [Tia95, Leh96, Hab90, Abe98] it is mentioned that characterizing workloads of real-time systems using *a priori* worst-case execution times can lead to poor resource utilization, particularly when the difference between WCET and normal execution time is large. It is stated in [Ste97, Abe98] that accurately measuring WCET is often difficult and sometimes impossible. In response to such difficulties, techniques for detection and handling of deadline violations have been developed [Jah95, Str97, Ste97]. Paradigms which generalize the execution time model have also been developed. Execution time is modeled as a *set* of discrete values in [Kuo97], as an interval in [Sun96], and as a probability distribution in [Leh96, Str97, Tia95, Atl98]. Most models consider execution time to apply to the job atomically; however, some paradigms [Liu91, Str97] view jobs as consisting of mandatory and optional portions; the mandatory portion has an *a priori* known execution time in [Liu91], and the optional portion has an *a priori* known execution time in [Str97]. Most of these approaches assume that the execution characteristics (set, interval or distribution) are known *a priori*. Others have taken a hybrid approach; for example, in [Hab90] *a priori* worst case execution times are used to perform scheduling, and a hardware monitor is used to measure *a posteriori* task execution times for achieving adaptive behavior. In [Bra98, Wel98], resource requirements are observed *a posteriori*, allowing applications which have not been characterized *a priori* to be accommodated. For applications with *a priori* characterizations, the observations are used to refine the *a priori* estimates. These characterizations are then used to drive dynamic resource allocation algorithms.

In this paper we present an approach that is appropriate for systems which experience large variations in workload. The approach dynamically manages a distributed collection of computing resources by continuously computing and assessing QoS and resource utilization metrics that are determined *a posteriori*.

Specifically, the paper presents an adaptive distributed system reference architecture that is suitable for such an approach.

The Distributed System Reference Architecture

Figure 1 depicts a distributed system reference architecture. This reference architecture provides the capabilities and infrastructure needed to construct multi-

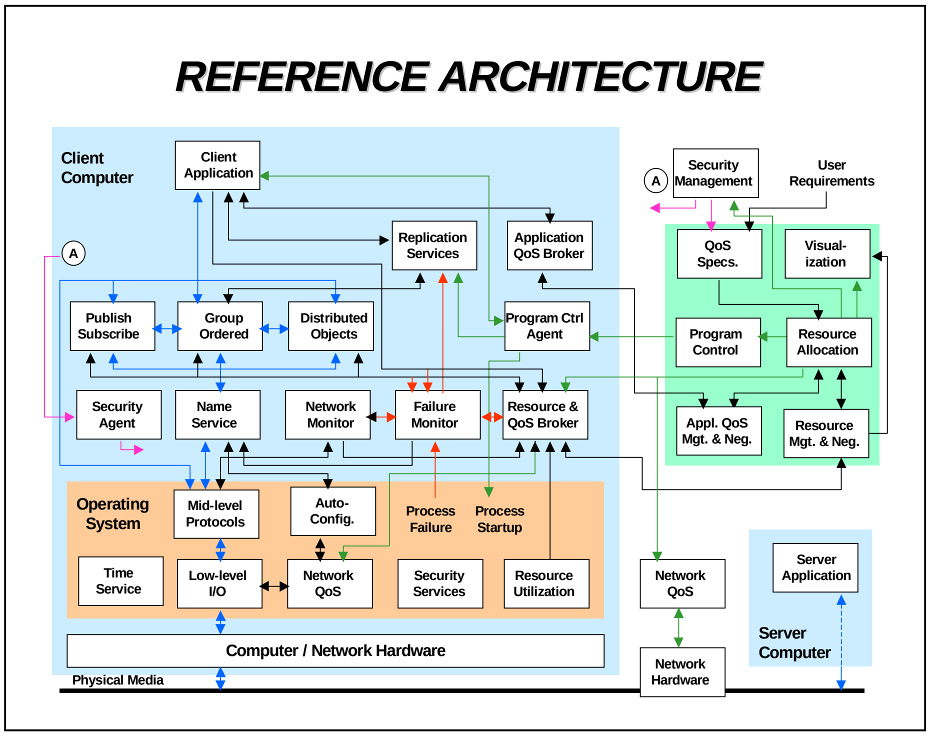

Figure 1. A distributed system reference architecture.

component, replicated, distributed object real-time systems that negotiate for a given level of QoS from the underlying distributed computing resources. The diagram shows the functional architecture structure needed to support distributed application programs for a computer containing a client application. This structure is repeated throughout the computers of the distributed system; in particular, the computer hosting the server application contains a comparable structure. Also executing somewhere in the distributed system's collection of computers is a set of QoS management components that interact with the computers, network components and applications of the distributed system to provide QoS management.

Besides the applications themselves, the components of the reference architecture consist of four primary types: operating systems, network services, high-level communication and state management middleware services, and resource management services. The operating system services are those needed to support real-time applications; operating system services are not discussed further herein. The network services provide low-level network communications and time management capabilities. The middleware components provide the ability to communicate in accord with three high level communication models: publish-subscribe; group programming, with associated ordered multicast and state data synchronization services; and distributed object programming. The QoS management services consist of computing resource and application status monitoring and reporting services, QoS negotiation services, and program control services.

Taken together, these services allow distributed applications to perform allocated processing functions, to communicate with each other, to perform fault detection and recovery activities, to perform load sharing activities, and to negotiate resource utilization and QoS requirements with the underlying distributed system infrastructure. This architecture has been partially implemented and successfully employed for dynamic management of QoS and distributed computing resources within a Navy distributed computing testbed. Features of the testbed include real-time mission critical computing, fault tolerance and scalability.

The components of the architecture are explained in the remainder of this document. Section 3 describes the *network* QoS reference architecture and introduces some additional details not shown in Figure 1. Section 4 explains the *system composition* reference architecture, and in section 4 the *resource and QoS* reference architecture is discussed.

The Network QoS Reference Architecture

The network provides a means of transferring information among the processors that compose a particular system. For the purposes of this Network QoS Reference Architecture (NQRA) there are two types of information passed through the network: (1) lower-level signaling and (2) system information. Lower-level signaling provides the means for network to operate and pass system information correctly. Examples of lower-level signaling are link operations, status and control information, routing information as well as the control information needed to support network QoS negotiation and QoS signaling for other Reference Architecture components. It is assumed that the necessary resources are allocated sufficiently to support the transfer of lower level signaling, as QoS control is exercised over the transfer of system information.

Some network provisioning decisions are performed outside of QoS control; for example, automatic fault recovery may support instantaneous reconfiguration of a host’s network connection. QoS control is applied to network fault recovery at a higher level, as the resource manager must consider which processors are using network connections which may have reconfigured, leaving no redundancy and thus no recovery capability for further failures.

Figure 2 depicts the subset of the distributed system reference architecture that is the NQRA. While the NQRA architecture is only shown in the client host, it is assumed that all hosts contain this architecture. Unshaded boxes are utilized by only the NQRA. Shaded boxes are used by the NQRA and the other Reference

Architecture Components. The components of the NQRA within a host computer include: 1) Security Agent: makes network security service requests on behalf of applications; 2) Security Services: provides multi-level network security services such as encryption, registration, and key management; 3) Name Service: provides dynamic logical to physical name translation for hosts and services; 4) Auto Configuration: allows network components to autonomously initialize and dynamically reconfigure; 5) Network Monitor: gathers remote performance metrics and failure notifications; 6) Failure Monitor: collects and acts upon network resource failure reports; 7) Resource & QoS Broker: correlates local/remote network performance metric/failure reports and submits QoS requests to the Network QoS component; gathers and correlates elementary monitoring information for determining system-level QoS and performance metrics; 8) Mid-level protocols: provides basic network communications services; 9) Time Services: provides accurate high-resolution global time; 10) Low-level I/O: underlying I/O and lower-level signaling mechanisms which support Network QoS component functionality (e.g. queue schedulers, packet classifiers, etc.); 11) Network QoS: dynamically allocates network resources to meet QoS requested by other components; 12) Resource Utilization: provides a local view of network resource utilization/performance metrics and failure notifications; also collects elementary host status and load metrics.

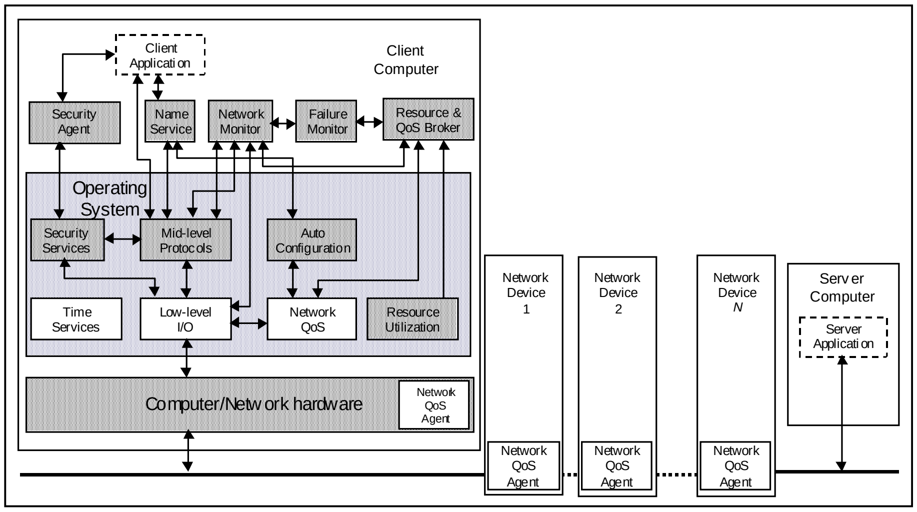

Figure 2. The network QoS reference architecture.

Within a network device (e.g. a switch, router, hub, etc. which is part of the fabric used to interconnect host computers) there is a single NQRA component, the Network QoS Agent which processes and acts upon lower-level signaling and provides status/performance reports back to host nodes which contain all of the NQRA components. The NQRA architecture is applicable to a variety of environments (e.g. LAN, WAN, mobile, etc.). The environment type is defined by the number and type of network devices along the QoS-enabled path that interconnects the communicating hosts.

As an example of a typical interaction of the NQRA components, suppose a communications channel with a given QoS requirement must be established between client and server applications on two different hosts. To communicate with the server, the client first uses the Name Service to find a physical address for it. A security association is established with the server via the security agent which uses the underlying security services component. The QoS requirements for the application are negotiated among the Network QoS Agents in each of the network devices required to establish a QoS-enabled path between the client and server. Finally, with the QoS-enabled path established, communications commences between the client and server applications. If a network device on the QoS enabled path is unable to meet the required QoS, the Network Monitor notifies the Failure Monitor and another QoS-enabled path is established by the Network QoS components and agents. All communications pass through the low-level I/O component as scheduled by the Network QoS component.

The System Composition Reference Architecture

System composition (SC) deals with services that allow for the composition of distributed, real-time, survivable applications. The SC components and their interactions are illustrated in Figure 3. The SC architecture components are: (1) Application: The unique portion of a given distributed process, the application implements specific algorithms of the combat system [Har94, Wel96], and is the driver of a subset of QoS requirements pertaining to real-time, survivability, security, and scalability. (2) Replication Service: Supports application management of replication for the local process, including load sharing, synchronization of replica state data, and replica initialization and termination handling. (3) Distributed Objects: An implementation of the CORBA standard with support for replicated objects, load sharing, and QoS. (4) Group Ordered Communications: One of several available middleware tools that provides a group programming model of interaction, ordered messages within a communications group, atomic message delivery, support for ordered state transfer, and ordered group membership change events; examples of such tools include Ensemble (Cornell), Isis (Stratus), RTCAST (U Mich), and Spread (JHU). (5) Data Distribution Service: Support for potentially high bandwidth, low latency multicast, such as might be provided by a publish/subscribe middleware tool; for example, Salamander (U Mich), RTI NDDS, or TIBCO Rendezvous (potentially with support by an underlying group programming tool). (6) Name Service: A system resource that provides a common, location-independent handle for referencing any other process or application in the distributed system. A hierarchical name reference is highly desirable; such that classes of applications can be referenced (e.g. all track distribution applications) as well as specific instances of an application. Although the name service is a system-level resource, node-level access is necessary to meet real-time service and survivability requirements. (7) Failure Monitor: Detects faults and failures of processes and host resources.

In a typical scenario, an application is initiated by the RM component. During initialization, RM provides initial replication and configuration information to support management of replica state data synchronization. Application security data is communicated via the security service, which in turn updates the system name service with the appropriate security status. A specification of the types of replication supported for different communications channels is provided to the local replication service. This information, combined with the current configuration data, is used by the replication service to initiate state transfer, state data synchronization, and load sharing allocations, as required, via the appropriate middleware services. The initial replica configuration for each of the communications channels (e.g. load sharing, primary/shadow) is reported to the application when all interfaces are initialized and stable. At this point, the application can begin its normal processing.

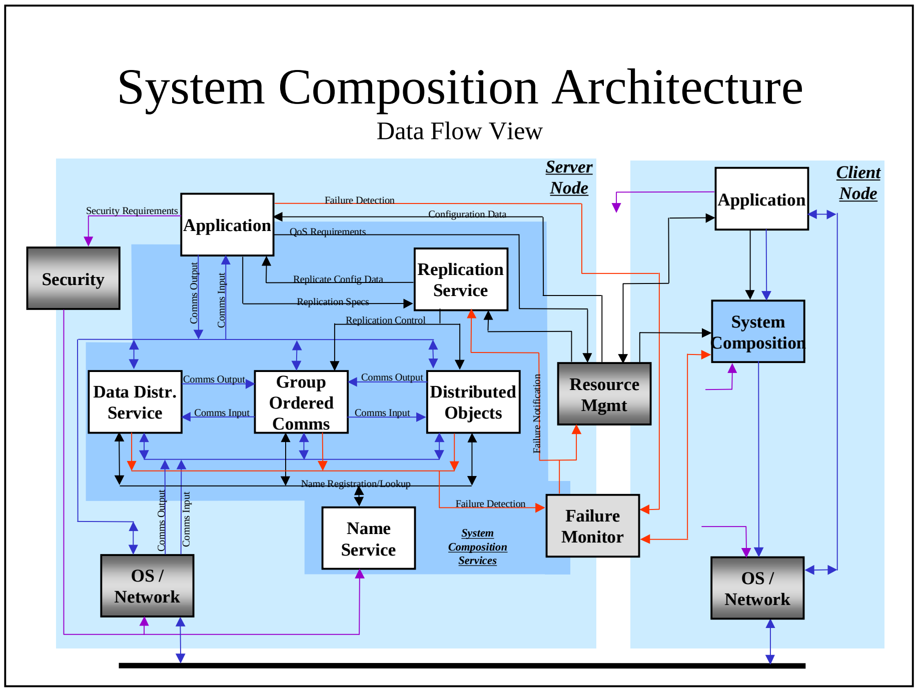

Figure 3. The system composition components and their interactions.

Application-to-application communication occurs via the appropriate middleware interface, either distributed object, group comms, data distribution, or, when necessary, UDP/TCP/IP. It is envisioned that the distributed object and data distribution middleware may eventually utilize group communications services where reliable, ordered, communications support is required. The system name service is used for common, logical, location-independent addressing, and security-level information is provided with each communication to assure that security constraints are not violated. During algorithm execution, the application communicates with the RM component, reporting its ability to meet its QoS requirements, with regard to processing and data latencies.

Application or communication errors detected at the application and middleware levels are report to the failure monitor. Errors occurring elsewhere in the distributed system that directly affect an application's replication or load sharing configuration, are reported by the failure monitor via the local replication service. Load redistribution and replica role changes are made in real-time by the replication service, based on predefined algorithms. Those decisions are reported back to the application. Reconfiguration changes made by RM are also reported to the application via the replication service, so that load redistribution and replica role decisions can be made, as appropriate.

The Resource and QoS Management Reference Architecture

Resource management (RM) and QoS management (QM) negotiate to allocate computing and communication resources in a way that attempts to satisfy all application QoS requirements. The distributed RM and QM reference architecture is shown in Figure 4, and consists of the following components: (1) QoS Specifications: Describe the static features of application software systems, and the distributed hardware and network systems; also describe QoS requirements pertaining to real-time, survivability, security and scalability. (2) Application QoS Broker: Collects elementary time-stamped events from applications and converts them into program-level QoS metrics. Also participates in QoS negotiation. (3) Application QoS Management & Negotiation: Monitors real-time path and application QoS and detects/predicts QoS violations; diagnoses local causes of QoS violations and suggests corrective reallocation actions to the Resource Management & Negotiation component; maintains application resource usage profiles; negotiates QoS with Resource Management & Negotiation component on behalf of managed applications; performs local stability analysis. (4) Resource Management & Negotiation: Oversees the resource management processes; performs global stability analysis; leads QoS negotiation in overloaded conditions. (5) Resource Allocation: Performs system level resource allocation tradeoff analysis via synthesis of (a) host and network utilization metrics and (b) tactical application performance and resource usage metrics. Performs global diagnosis to determine the causes of QoS violations and system faults, analyzes potential corrective actions, and selects specific hardware and software reconfiguration actions. (6) Program Control: Implements application control actions (startup, shutdown, reconfiguration) requested by the Resource Allocation components or by an operator, and automates the control process (e.g., analyzing and handling startup dependencies, configuration of command line arguments, etc...). (7) Program Control Agent: Provides the low-level mechanisms for starting, stopping, and reconfiguring application programs. (8) Visualization: Displays information regarding system configuration and performance; status and QoS performance of the applications, middleware, hosts, and networks; and details of the reallocation decision process.

A general interaction scenario among the architecture components which are directly related to resource and QoS management occurs as follows. The user QoS requirements for the application system are described in terms of the System/QoS Specification language. Specifications are also produced for the software system structure, the software startup and shutdown procedures, and the distributed hardware system. The specifications are used by the Resource Allocation components to initially allocate the applications of the software system to the distributed hardware resources. The initial allocation is enacted by Program Control, which notifies Program Control Agents on each host to start the appropriate applications. Once operational, the distributed applications interact as needed via the Publish/Subscribe, Group Ordered and/or Distributed Objects communication protocols. Application QoS, performance, and statuses are monitored via the Application QoS Broker components, and correlated and analyzed for actual or predicted QoS violations by the Application QoS Management & Negotiation component. Hardware utilization levels and program statuses are monitored by the Resource Utilization and Failure Monitor components. When inadequate QoS, hardware saturation or failure, or abnormal program termination is detected (or predicted), the Resource Allocation and the Resource Management & Negotiation components subsequently select a better allocation of resources and issue requests to the Program Control components to enact the new allocation. In situations where inadequate resources do not exist, QoS negotiation is performed via iteration among Resource Allocation, Resource Management and Negotiation, Application QoS Management & Negotiation, Application QoS Broker and the Client Application.

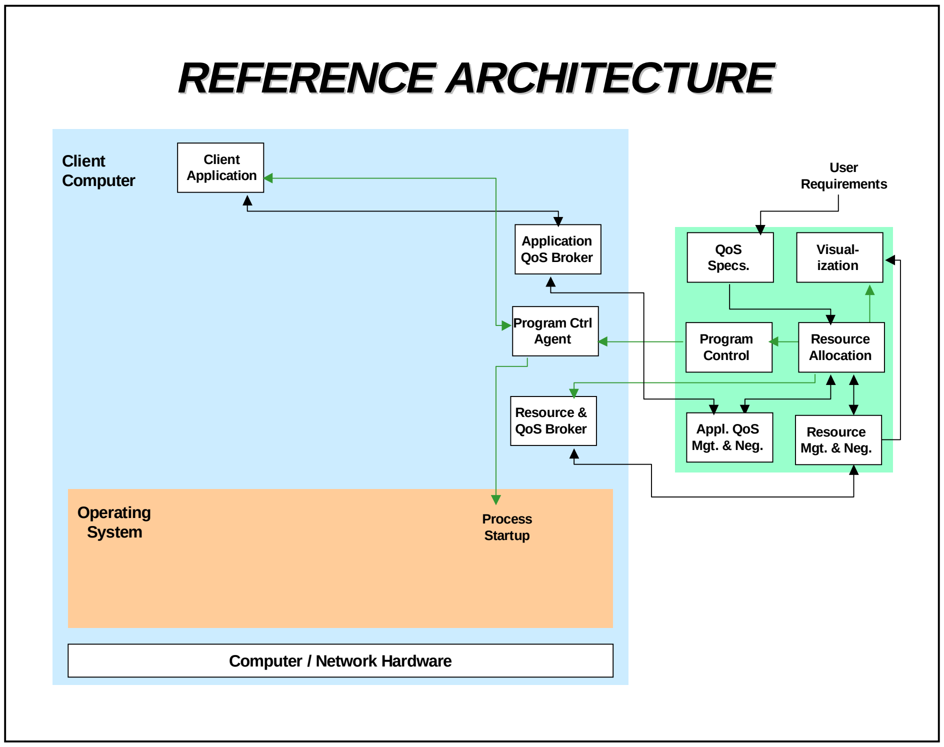

Figure 4. A distributed QoS and resource management reference architecture.

Conclusions and Ongoing Work

This paper presents adaptive QoS and resource management technology for distributed real-time systems with *a-posteriori*-determined workloads. A reference architecture that provides a plan for adaptive QoS and resource management technology within this context was presented. This reference architecture provides the capabilities and infrastructure needed to construct multi-component, replicated, distributed object real-time systems that negotiate for a given level of QoS from the underlying distributed computing resources.

A significant portion of the reference architecture has been realized. As reported in [Wel99], the effectiveness of our approach was demonstrated within an experimental large-scale combat system. The results show that real-time QoS violations were successfully detected, appropriate recovery actions were performed, and required QoS was restored. In addition, we tested the ability to provide survivability services to real-time application systems in a timely manner. Across all tests, the total response time of QoS and resource management services was less than one second, even under heavy tactical and system loads.

Ongoing work includes the realization of additional portions of the architecture, and the experimental assessment and comparison of candidate components in specific portions of the architecture. The experimental assessment efforts include the development of computing and communication benchmarks which exhibit characteristics of dynamic real-time systems.

References

[Abe98] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. of the 19th IEEE Real-Time Sys. Symposium*, 3-13, IEEE Computer Society Press, 1998.

[Atl98] A. Atlas and A. Bestavros, "Statistical rate monotonic scheduling," in *Proceedings of the 19th IEEE Real-Time Systems Symposium,* 123-132, IEEE Computer Society Press, 1998.

[Bra98] S. Brandt, G. Nutt, T. Berk and J. Mankovich, "A dynamic quality of service middleware agent for mediating application resource usage," in *Proceedings of the 19th IEEE Real-Time Sys. Symposium,* 307-317, IEEE Computer Society Press, 1998.

[Hab90] D. Haban and K.G. Shin, "Applications of real-time monitoring for scheduling tasks with random execution times ," IEEE Trans. on Software Engineering, **16(12)**, December 1990, 1374-1389.

[Har94] Robert D. Harrison Jr., "Combat system prerequisites on supercomputer performance analysis," in *Proceedings of the NATO Advanced Study Institute on Real Time Computing*, NATO ASI Series **F(127)**, 512-513, Springer-Verlag 1994.

[Jah95] F. Jahanian, "Run-time monitoring of real-time systems," in *Advances in Real-time Systems,* Prentice-Hall, 1995, 435-460, edited by S.H. Son.

[Kuo97] T.E. Kuo and A. K. Mok, "Incremental reconfiguration and load adjustment in adaptive real-time systems," *IEEE Transactions on Computers,* **46(12)**, December 1997, 1313-1324.

[Leh96] J. Lehoczky, "Real-time queueing theory," in *Proceedings of the 17th IEEE Real-Time Systems Symposium*, 186-195, IEEE Computer Society Press, 1996.

[Liu73] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM,* **20**, 1973, 46-61.

[Liu91] J.W.S. Liu, K.J. Lin, W.K. Shih, A.C. Yu, J.Y. Chung and W. Zhao, "Algorithms for scheduling imprecise computations," *IEEE Computer,* **24(5)**, May 1991, 129-139.

[Sha91] L. Sha, M. H. Klein, and J.B. Goodenough, "Rate monotonic analysis for real-time systems," in *Sched. and Res. Mgmt.,* Kluwer, 1991, 129-156, edited by A. M. van Tilborg and G. M. Koob.

[Ste97] D.B. Stewart and P.K. Khosla, "Mechanisms for detecting and handling timing errors," *Communications of the ACM,* **40(1)**, January 1997,87-93.

[Str97] H. Streich and M. Gergeleit, "On the design of a dynamic distributed real-time environment," in *Proceedings of the 5th International Workshop on Parallel and Distributed Real-Time Systems*, 251-256, IEEE Computer Society Press, 1997.

[Sun96] J. Sun and J.W.S. Liu, "Bounding completion times of jobs with arbitrary release times and variable execution times," in *Proceedings of the 17th IEEE Real-Time Systems Symposium,* 2-11, IEEE Computer Society Press, 1996.

[Tia95] T.S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.C. Wu and J.W.S. Liu, "Probabilistic performance guarantee for real-time tasks with varying computation times," in *Proceedings of the 1st IEEE Real-Time Technology and Applications Symposium,* 164-173, IEEE Computer Society Press, 1995.

[Web] *Webster's New World Dictionary of the American Language (College Edition)*, The World Publishing Company.

[Wel95] L. R. Welch, A. D. Stoyenko, and T. J. Marlowe, "Modeling resource contention among distributed periodic processes specified in CaRT-Spec," *Control Engineering Practice*, **3(5)**, May 1995, 651-664.

[Wel96] L. R. Welch, Binoy Ravindran, Robert D. Harrison, Leslie Madden, Michael W. Masters and Wayne Mills, "Challenges in Engineering Distributed Shipboard Control Systems," *The IEEE Real-Time Systems Symposium (in Proceedings of the Work-in-Progress Session)*, December 1996, 19-22.

[Wel98] 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.

[Wel99] L. R. Welch, Paul V. Werme , Larry A. Fontenot, Michael W. Masters, Behrooz A. Shirazi, Binoy Ravindran and D. Wayne Mills, "Adaptive QoS and Resource Management Using A Posteriori Workload Characterizations," *Technical Report,* The University of Texas at Arlington, Computer Science and Engineering Department, 1999.