DynBench: A Dynamic Benchmark Suite for Distributed Real-time Systems [1]

Behrooz Shirazi[2], Lonnie Welch[2], Binoy Ravindran[3], Charles Cavanaugh[2], Bharath Yanamula[2], Russ Brucks[2], Eui-nam Huh[2]

[2]University of Texas at Arlington
Department of Computer Science and Engineering
Box 19015
Arlington, TX 76019-0015
{shirazi|welch}@cse.uta.edu
[3]Virginia Tech
The Bradley Dept. of Electrical and Computer Engineering
340 Whittemore Hall (Mail Code 0111)
Blacksburg, VA 24061

Abstract. In this paper we present the architecture and framework for a benchmark suite that has been developed as part of the DeSiDeRaTa project. The proposed benchmark suite is representative of the emerging generation of distributed, mission-critical, real-time control systems that operate in dynamic environments. Systems that operate in such 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 be very difficult to characterize statically, even by time-invariant statistical distributions. The proposed benchmark suite (called DynBench) is useful for evaluation of the Quality of Service (QoS) management and/or Resource Management (RM) services in distributed real-time systems. As such, DynBench includes a set of performance metrics for the evaluation of the QoS and RM technologies in dynamic distributed real-time systems. The paper demonstrates the successful application of DynBench in evaluation of the DeSiDeRaTa QoS management middle-ware.

Introduction and Background

Fig. 1 [Koob98] shows a generic *Quality of Service* (QoS) management architecture for distributed real-time systems. The distributed real-time applications are supported by a set of translucent systems layers that provide communication libraries, group communication facilities, and other support routines. Additionally, the system provides a set of middle-ware components to support QoS negotiation and Resource Management (RM). A typical application requires a set of QoS parameters such as real-time deadlines and fault-tolerance capabilities. On the other hand, the

[1] 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.

system at any given time can provide some level of QoS for the applications, given its available resources. Thus, the QoS negotiation components need to detect the available system resources and reconcile the applications' requirements with the system's resources. The resource management component is then deployed to re-allocate the resources to the applications according to the negotiated QoS.

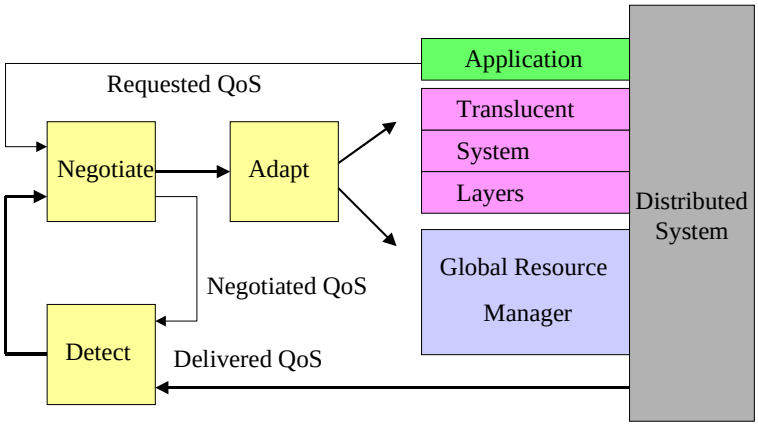

Fig. 1. A generic QoS management architecture for distributed system.

The behavior of a typical distributed real-time system application, shown in Fig. 1, may be dynamic in at least 3 dimensions: (1) the application's input data size and event arrival rates may be unknown; (2) the application may be scalable and fault-tolerant, implying that the number of program components at execution time can vary; and, (3) the application behavior may be influenced by independent events, such as load of a host or network. There are a number of QoS negotiation and RM technologies [H+97, CS+97, DL97, Hon97, RLLS97, RSYJ97, ZBS97] that are deployed in distributed real-time systems. However, there are no standard benchmarks, with the outlined dynamic behaviors, that can be used to evaluate such technologies. The goal of this work is to address this void. In this paper we present (i) the design and development of a benchmark suite and the framework to control the dynamic behavior of the benchmark in a repeatable way, and (ii) the definition and implementation of performance assessment metrics for evaluation of QoS and RM technologies.

To validate and assess QoS negotiation and resource management middle-ware in distributed real-time systems, there needs to be a *generic*, *repeatable*, and *deterministic* experimental setup. As part of the DeSiDeRaTa (Dynamic, Scalable, Dependable, Real-Time systems) project, we developed *DynBench*, a Dynamic real-time Benchmark suite. DeSiDeRaTa provides the QoS negotiation and RM middle-ware needed to address the needs of dynamic distributed real-time systems [Ravin98a-b, Welch97a, Welch98a-d].

For distributed real-time systems that have strong QoS requirements, a benchmark must have the following properties: (i) scalability, (ii) allowance of varying input rates for unknown environments, (iii) consideration of the unpredictable environment (host or network) behavior, and (iv) provision of facilities for easy collection of

metrics from heterogeneous system architectures and networks. DynBench has been designed to have all of the above characteristics.

Traditional computing benchmarks (e.g., Dhrystone [Wei84], Whetstone [Cur76], AIM[Pri89], Linpack [Wei90], Livermore Loops, NAS Kernels, Fhourstones) are inadequate for characterizing dynamic distributed real-time systems. Primarily, they were not designed to exhibit behavior characteristic of control systems, such as periodic, transient and transient periodic activation/deactivation. Also, they focus on component level benchmarking; thus, they do not provide the notion of large grain entities with end-to-end timing constraints. The Rhealstone benchmark [Kar89] also has a small grain focus, but it focuses on system features that are relevant to real-time systems. It measures task switch time, preemption time, interrupt latency time, inter-task message latency, semaphore shuffle time, and deadlock-break time for *components* of real-time systems.

The Hartstone benchmark [Wei89, Kam91, Wei92, Ujv97] builds on the traditional component level synthetic benchmarks by incorporating them into a framework that provides periodic and aperiodic activation of distributed, time constrained communicating tasks. The authors show how to use the benchmark to evaluate system software (real-time O.S. and real-time language run-time system), host hardware, communication system software, node communication hardware, and physical channel characteristics and protocols. The primary use of Hartstone is to determine the workloads under which these various components of the real-time platform "break down"; this is done by incrementally increasing workload (sizes of messages, number of tasks sending/receiving messages, number of messages) until a real-time QoS violation occurs.

Like Hartstone, SWSL [Kis94] and SW [Kis96] provide distributed, communicating periodic and aperiodic tasks that use synthetic workloads. However, the authors provide a more comprehensive framework than in [Wei89, Kam91, Wei92, Ujv97]. Workloads are described using SWSL (synthetic workload specification language) and then compiled by SWG (synthetic workload generator) to produce executables for all processors in a distributed or parallel system. SWSL specifications are compact and flexible, employing data flow graphs at the task level and control constructs and synthetic operations at the operation level. To allow rapid specification of large concurrent systems, SWSL allows instantiation of parameterized objects, and specification of the task-to-processor mapping. Stochastic workloads are generated at run-time via random number generators with specific distributions and parameters. (SWSL provides a library of random number generators with various distributions.) To reduce the time required to perform multiple experiments, SWSL supports a multiple run facility, which uses a set of static parameters to drive a set of test runs.

While there has been significant work performed in benchmarking and synthetic workload generation for real-time systems, DynBench addresses some important issues that have not been dealt with previously. We extend the workload models of Hartstone and SW/SWSL beyond periodics and aperiodics, by including transient periodics (or quasi-continuous paths [Wel98a]). To test dynamic QoS and resource management technology, DynBench applications are dynamically controllable and dynamically relocatable; furthermore, they are dynamically replicable (for survivability and scalability), so the number of replicas can vary arbitrarily at run-time under the control of QoS and resource managers. Previous work did not support dynamic adaptation: in Hartstone, a QoS violation causes the test to terminate; in

SWSL, objects can be replicated, but the number of software components as well as the locations where they execute are fixed statically. The variations in workload can be stochastic in SWSL; in Hartstone, workload can be increased incrementally until a QoS violation occurs. DynBench permits workload to increase and to decrease dynamically, according to online user input, and according to user–defined experiment scenario files; these workload variations can be fully dynamic, i.e., they need not be constrained by time-invariant statistical distributions. Another obvious distinction of DynBench is that it employs real workloads (with features similar to those described in [Dav80, Mol90, Kav92, Har94, Wel96]), instead of synthetic workloads, avoiding the problems of (1) obtaining a realistic synthetic workload and (2) accurately characterizing a realistic workload and its low-level data-dependent behaviors. Also, this has the advantage of allowing a user to "see" that the appropriate QoS is being delivered to the benchmark application system by observing its tactical behavior. (In the C3I Parallel Benchmark [Met96], real workloads are also employed for seven subsystems: terrain masking, hypothesis testing, route optimization, plot-track assignment, synthetic aperture radar imaging, threat analysis, decision support systems and map-image correlation. However, due to their classified nature, the real-time versions of those benchmarks are only available to U.S. government employees and their contractors.) DynBench also provides instrumentation support necessary for resource and QoS management, a feature not found in previous work. Additionally, we have used the "computing cornerstones" presented in [Mast95] as a basis for defining a set of metrics for assessing QoS and resource management technology; and we provide an environment that automatically gathers the metrics.

DynBench Framework

Fig. 2 illustrates the DynBench framework. The DynBench framework controls the dynamic behavior of the benchmark in a repeatable way and collects performance assessment metrics for evaluation of QoS and RM technologies. The lower four components (benchmark, experiment generator, program control, and monitor) comprise the DynBench benchmark suite. The benchmark suite is used to evaluate any QoS/RM technology, represented by the QoS/RM services components.

The *benchmark* component is a distributed real-time control system designed to behave in a dynamic manner. Application instrumentation data collected from the benchmark goes to the *monitor* component, which calculates QoS metrics from the data. The monitor also takes instrumentation data from the QoS/RM services and calculates assessment metrics, storing them in the *assessment metrics file*. The *QoS/RM services* components represent the middle-ware components that perform QoS negotiation and resource management based on the QoS metrics provided by the monitor. The *experiment generator* allows the user to create multiple instances of the benchmark application. This also allows the user to dynamically change the characteristic behavior of the benchmark system at run-time. The *meta spec* file is provided to the experiment generator for creating different instances of the benchmark. The *commands file* allows the user to dynamically change the behavioral characteristics of the benchmark systems and that of the hardware platform in a deterministic and repeatable manner. The *spec file* describes the software QoS requirements as well as the hosts and networks that make up the distributed system.

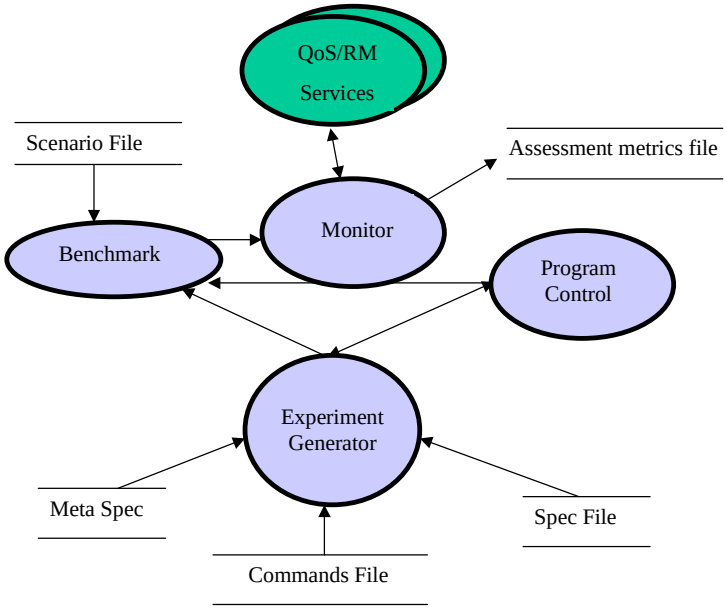

Fig. 2. The DynBench framework.

Benchmark:

The dynamic path paradigm [Wel96, Welch97a] provides a convenient abstraction for capturing the characteristics of dynamic real-time systems. In this paradigm, a path consists of a set of large-grain communicating programs with end-to-end QoS requirements. Fig. 3 depicts a typical real-time control system, such as an anti-air defense, a robotics, or an autonomous vehicle system. Control systems perform actions in response to conditions detected in the environment that they inhabit. They typically consist of sensors that monitor external conditions, and a situation assessment path that filters, correlates and evaluates the sensor data. Events detected by the situation assessment path activate the engagement path that plans and initiates responses via actuators. This in turn activates the monitor and guide path that guides the actions to successful completion. The processing performed by a control system depends on the environment in several ways. The load of the situation assessment path at any point in time is a function of the number of environmental elements detected by the sensors. The loads of the engagement path and the monitor and guide path depend on the rate at which environmental events occur. Thus, the characteristics of a control system's environment are important engineering considerations, making such a system ideal as a dynamic benchmark for distributed real-time systems.

Using the *dynamic path* paradigm, the DynBench benchmark application is modeled after typical distributed real-time military applications such as an air defense subsystem. Fig. 4 shows the three *dynamic paths* from the DynBench benchmark application. The *detect* path (path 1) is a continuous path that performs the role of

examining radar sensor data (radar tracks) and detecting potential threats to a defended entity. The sensor data are filtered by software and are passed to two evaluation components, one is software and the other is a human operator. The detection may be performed manually, automatically, or semi-automatically (automatic detection with manual approval of engagement recommendation). When a threat is detected and confirmed, the transient *engage* path (path 2) is activated, resulting in the firing of a missile to engage the threat. After a missile is in flight, the quasi-continuous *guide missile* path (path 3) uses sensor data to track the threat, and issues guidance commands to the missile. The *guide missile* path involves sensor hardware, software for filtering/sensing, software for evaluating and deciding, software for acting, and actuator hardware.

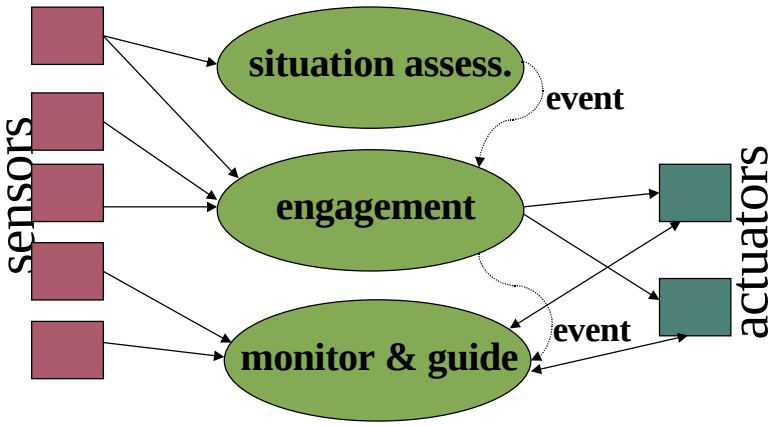

Fig. 3. A typical path-based real-time control system application.

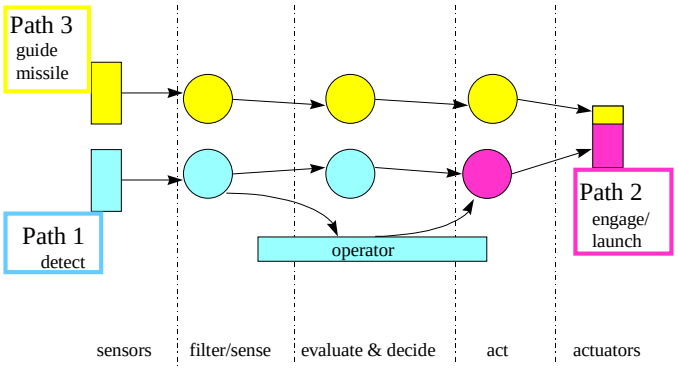

Fig. 4. The DynBench dynamic paths.

The *real-time* characteristics of the three paths are as follows. The continuous *detect* path is a data-driven path, with desired end-to-end cycle latencies for

evaluating radar track data and detecting potential threats. The sensor data are provided to the path periodically, but the periodicity can change dynamically. For example, if a potential threat is detected, the frequency of sensing may increase.

The transient *engage* path is a sporadic, high-priority, "one-time" path. It is activated by an event from the *detect* path. Each time the *engage* path is activated, it executes only once. There is an end-to-end QoS timeliness objective on the *engage* path that typically has a higher priority than the timeliness QoS of the *detect* path or the *guide missile* path.

The *guide missile* path is transient periodic (or quasi-continuous) in the sense that it is sporadically activated and deactivated; but when active, it behaves like a continuous path. It is activated by the *missile launch* event. Once activated, the path continuously issues guidance commands to the missile until it detonates. Thus, the guide missile path is an example of a quasi-continuous path. The required completion time for one iteration is dynamically determined based on characteristics of the threat. For example, a fast moving, nearby threat will require issuing more frequent guidance commands than a slower-moving threat that is farther away.

QoS Specification

The DeSiDeRaTa environment includes a specification language called D-Spec [Welch98a] that is used for specification of QoS requirements of distributed systems, such as DynBench, that operate in unknown environments. D-Spec is useful for describing real-time QoS requirements in terms of end-to-end paths through application programs.

Experiment Generator:

In addition to the benchmark application described in the previous section, the DynBench framework includes an experiment scenario generator, which allows construction of multiple instances of the benchmark application using a meta spec file. Further, it allows dynamic changes to the behavioral characteristics of the benchmark systems and that of the hardware platform in a deterministic and repeatable manner using a commands file. The commands file contains a description of a set of events as a function of time. These events include increasing/decreasing the tactical load, killing a program or a host, and increasing/decreasing the load of a host or a network subsystem. The generator interfaces with the program control and resource manager components of the middle-ware, and the benchmark.

Monitor and Program Control:

The Quality of Service (QoS) being received by the DynBench benchmark applications is constantly monitored and passed to the QoS and RM services at their path and subpath levels of abstraction. The path and program latencies are currently the main QoS objectives monitored by the QoS monitor. Similarly, the monitor gets performance data from the QoS/RM services (such as latency of RM operations), computes QoS services assessment metrics, and outputs the metrics to the assessment metrics file. There is a QoS Monitor for each path in the subsystem. The main

objective of the Program Control is to communicate a program failure to the RM services or start execution of a program on a host as specified by the Resource Manager.

QoS Assessment Metrics

In order to evaluate different QoS and RM services in distributed real-time systems, there is a need for a set of standard performance metrics. In this section we define the desirable attributes of such performance metrics. Some of these metrics are already implemented in the DeSiDeRaTa middle-ware services using the DynBench application as a testbed. Other metrics are in the process of being implemented. We propose the following QoS/RM services assessment metrics [Mast95]:

- *Performance*
  Performance metrics for QoS/RM services include latencies of different components, whether or not the latencies are bounded and deterministic, complexity of the QoS/RM components, and overhead of the QoS/RM operations.
- *Accuracy*
  Another set of assessment metrics include the degree of accuracy of the QoS/RM services operations in terms of monitoring, detecting, predicting (future trends), and diagnosing application ills.
- *Capacity / Scalability*
  QoS/RM services can be distinguished by maximum capacities for handling faults, tactical load, software components, and hardware components.
- *Quality*
  This is probably one of the most important (as well as one of the most difficult to measure) assessment metrics for evaluation of QoS/RM services. A quality metric may include QoS violation rate, ratio of QoS to resources consumed, stability of the QoS/RM services, and sensitivity of a chosen allocation to: hardware faults, increased tactical load, or increased number of programs.
- *Survivability*
  Is the QoS/RM service itself fault-tolerant and survivable?
- *Openness*
  Does the QoS/RM service have an open architecture, allowing different components to be mixed and matched?
- *Intrusiveness*
  This metric measures the degree of intrusiveness of the QoS/RM service on the design, development and fielding of applications.
- *Testability / Verifiability*
  Is the QoS/RM service easily testable and verifiable?

Experimental Results

We have used the DeSiDeRaTa [Ravin98a-b, Welch97a, Welch98a-d] middle-ware testbed to conduct different experiments with the DynBench benchmark suite. The experimental system parameters and environment are summarized as follows. The distributed network of hosts consists 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 four hosts are connected via a 10/100MB per second Ethernet network. The HCI runs under Windows NT on a Pentium Class machine. The DynBench application consists of the situation assessment path with a latency deadline of 1 second.

This section presents scenarios showing system behavior for the DynBench situation assessment path under 3 different run-time conditions. In two scenarios, the DeSiDeRaTa adaptive resource management middle-ware monitors the real-time QoS of the DynBench application, detects QoS violations, diagnoses cause(s) of poor QoS, and recovers accordingly. In these experiments, the QoS is measured along one dimension, the latency of a path. Experiments that consider survivability QoS are also performed. Throughout these experiments we demonstrate how DynBench can be used to evaluate adaptive resource allocation technology in a dynamic real-time client-server-based system.

In the first test, we examined the scalability capabilities of DynBench and the DeSiDeRaTa middle-ware. The idea is that if we fix all DynBench and system parameters, but increase the tactical load (number of sensed tracks), then the middle-ware should detect this condition and scale up the DynBench program(s) that is the bottleneck. Fig. 5 depicts the results of this experiment. Here the observed situation assessment path latency is shown as a function of the DynBench cycles. The situation assessment path latency is set to 1 Sec. (1000 milli-sec) and is shown as a straight line. Initially, the Sensor, and Evaluate and Decide (ED) programs run on *nujersy*, Filter Manager (FM) and Evaluate and Decide Manger (EDM) run on *virginia*, and one copy of the Filter (FL) runs on *texas*. The middle-ware components are distributed among the four hosts.

During the first 26 cycles, the tactical load (number of sensed tracks) is set to 1000 and the path is running with latencies below the threshold. At cycle 27, using the experiment generator component of the DynBench, we increased the tactical load by 500. Note that the path latency increases above the threshold because of this dynamic change in the input data set size. The QoS Monitor component of the DeSiDeRaTa middle-ware detects the QoS violation during the [27, 43] cycle interval. In order to avoid threshing and filter out the noise from the observed QoS parameters, the QoS manger is configured to diagnose a QoS violation if there are 15 violations in any consecutive 20-cycle window. Once the QoS violation was diagnosed in cycle 43, another copy of the FL and ED programs were initiated in *desidrta*, causing the observed latency to fall below the threshold in the next cycle. This experiment shows that DynBench is clearly scalable and such a capability can be used to evaluate the scalability of distributed real-time middle-ware.

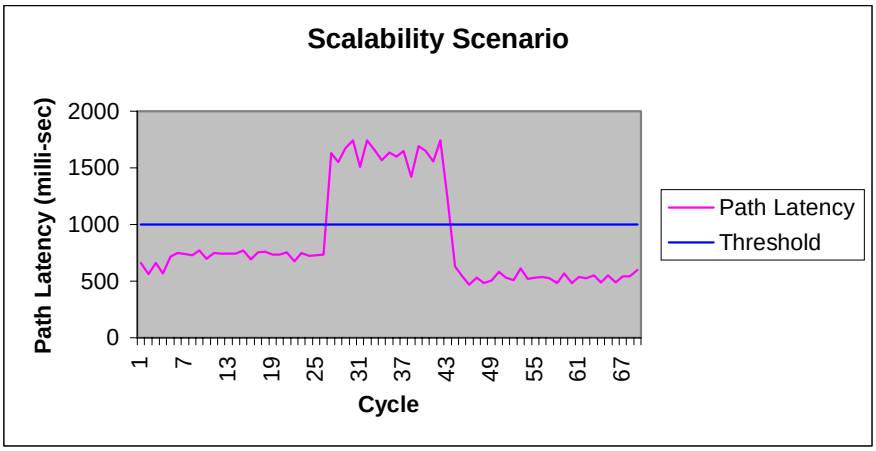

Fig. 5. The situation assessment path latency in the scalability test scenario.

In the second experiment, we examined the adaptability capability of the DeSiDeRaTa middleware. The idea is that if we fix all DynBench and system parameters, but increase the load of a host, then the middle-ware should detect this condition and perhaps move some of the DynBench programs to less loaded computing resources. Fig. 6 depicts the results of this experiment. Initially, FM and EDM programs run on *nujersy*, ED runs on *virginia*, and Sensor and FL run on *desidrta*.

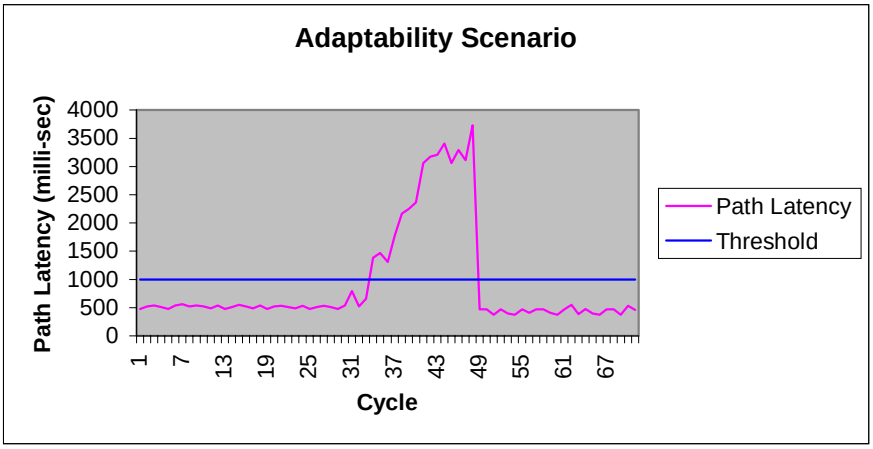

Fig. 6. The situation assessment path latency in the adaptability test scenario.

During the first 33 cycles, the tactical load (number of sensed tracks) is set to 1000 and the path is running with latencies below the threshold. At cycle 34, using the experiment generator component of the DynBench, we increase the load of the *desidrta* host by running 5 copies of a 500x500 matrix multiply application. Note that the path latency increases above the threshold because of this dynamic change in the

system resources. The QoS Monitor component of the DeSiDeRaTa middle-ware detects the QoS violation during the [34, 49] cycle interval. Once the QoS violation is diagnosed, the DeSiDeRaTa adaptive RM moves the FL program from *desidrta* to *nujersy* and moves the EDM program from *nujersy* to *virginia*. These moves cause the observed latency to fall below the threshold in the next cycle. This experiment shows that DynBench's experiment generator can inject dynamic events (such as increasing a host's load) during a distributed real-time application's execution; and such a capability can be used to evaluate the adaptability of QoS and resource management services in distributed real-time systems.

In the third experiment, we evaluated the survivability capability of the DeSiDeRaTa middle-ware in terms of detecting faulted application programs and restarting them. The idea is that if we use DynBench's experiment generator to kill an application program, then the middle-ware should detect this condition and restart the application. Fig. 7 presents the results of this experiment. Initially, the Sensor, FM, ED and EDM programs run on *desidrta* and FL runs on *nujersy*.

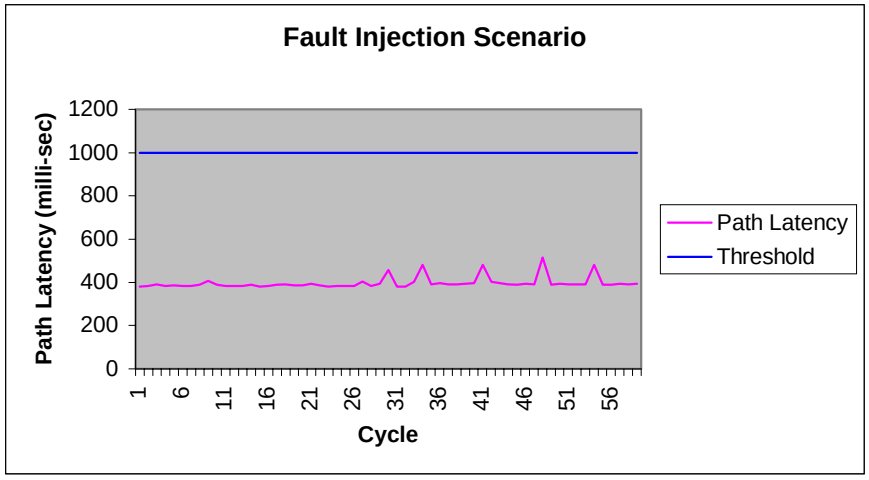

Fig. 7. The situation assessment path latency in the survivability test scenario.

At cycle 27, using the experiment generator component of the DynBench, we killed the ED program on *desidrta*. The QoS Monitor component of the DeSiDeRaTa middle-ware detects this condition in cycle 27 and restarts the ED on *virginia* in the next cycle. DynBench programs are state-less applications; thus, the recovery process is very quick. Once again, this experiment demonstrates the experiment generator's ability to dynamically inject events (such as killing a program) during an application's execution. Such a feature can be used to evaluate the fault detection and resolution capability of QoS and resource management services in distributed real-time systems.

QoS Assessment Metrics Results

Currently DynBench is set up to collect and report the *latencies* of the QoS and resource management services. Table 1 depicts the average latencies of a number of DeSiDeRaTa service components over multiple runs of the middle-ware with

DynBench. It is quite clear that in case of DeSiDeRaTa, all the QoS and resource management services take place in low hundreds of milli–sec., and the QoS violation detection and decision-making process take place in less than 10 milli–sec.

Table 1. DeSiDeRaTa QoS assessment metrics.

Metric	Average Latency (in milli-sec.)
QoS violation detection	1
RM decision-making process	3
RM best host selection for allocation	120
Total RM response time	123
Application start-up time (network dependent)	213

To measure the intrusiveness of the middle-ware, a series of experiments were conducted in which the real-time QoS of paths were measured in isolated and non-isolated conditions. The results show that the average maximal and minimal intrusiveness have been determined to be 3.47% and 0.299% of the path latency, respectively. Due to space limitations we are not able to present the details of these experiments in this extended abstract.

Conclusion

In this paper we presented the architecture and framework for a benchmark suite, called DynBench, that has been developed as part of the DeSiDeRaTa project. DynBench is one of the first benchmarking efforts representing "*dynamic*" real-time control systems. The benchmark suite is representative of the emerging generation of distributed, mission-critical, real-time control systems which operate in dynamic environments. The DynBench experiment generator framework takes into account the dynamic behavior of the application as well as the distributed system on which it runs.

DynBench was shown to be useful for evaluation of the quality of service management and/or resource management services in distributed real-time systems. We presented experimental results that validated DynBench as a credible benchmark for distributed real-time systems. As such, DynBench includes a set of performance metrics for the evaluation of the QoS and RM technologies in dynamic distributed real-time systems. The paper demonstrated the successful application of DynBench in evaluation of DeSiDeRaTa QoS management middle-ware.

References

[CS+97] S. Chatterjee, J. Sydir, B. Sabata, and T. Lawrence, "Modeling Applications for Adaptive QoS-based Resource Management," 2nd

IEEE High-Assurance System Engineering Workshop (HASE97), Bethesda, Maryland, August 1997.

[Cur76] H.J. Curnow and B.A. Wichmann, "A synthetic benchmark," *Computer Journal* 19(1), January 1976, 43-49.

[Dav80] C.G. Davis, "Ballistic missile defense: a supercomputer challenge," *IEEE Computer*, 30, 1980, 37-46.

[DL97] Z. Deng and J. W.-S. Liu, "Scheduling real-time applications in an open environment," *Real-Time Systems Symposium*, San Francisco, California, December 1997, pp. 308-319.

[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.

[Hon97] Honeywell Technology, "Adaptive Computing systems Benchmark," http://www.htc.honeywell.com/projects/acsbench/.

[H+97] J. Huang, etal., "RT-ARM: A real-time adaptive resource management system for distributed mission-critical applications", Workshop on Middleware for *Distributed Real-Time Systems*, RTSS-97.

[Kam91] N.I. Kamenoff and N.H. Weiderman, "Hartstone distributed benchmark: requirements and definitions," in *Proceedings of the 12th IEEE Real-Time Systems Symposium*, 1991, 199-208.

[Kar89] Kar, R.P. and Porter, K., "Rhealstone -- A Real Time Benchmarking Proposal," *Dr. Dobbs Journal* 14(2):4-24, February, 1989

[Kav92] K.M. Kavi and S.M. Yang, "Real-time systems design methodologies: an introduction and a survey," *The Journal of Systems and Software*, April 1992, 85-99, Elsevier Science Publishers.

[Kis94] D.L.Kiskis and K.G. Shin, "SWSL: A synthetic workload specification language for real-time systems," *IEEE Transactions on Software Engineering*, 20(10), Oct. 1994, 798-811.

[Kis96] D.L.Kiskis and K.G. Shin, "A synthetic workload for a distributed real-time system," *Journal of Real-Time Systems*,11(1), July 1996, 5-18.

[Koob98] G. Koob, "DARPA, Quorum mission statement," http://www.darpa.mil/ito/ research/quorum/index.html.

[Mast95] Michael W. Masters, "Real-time Computing Cornerstones: A System Engineer's View," Proceedings of the 3rd *Workshop on Parallel and Distributed Real-time Systems*, 1995

[Met96] R.C. Metzger, B. VanVoorst, L.S. Pires, R. Jha, W. Au, M. Amin, D.A. Castanon and V. Kumar, "C3I parallel benchmark suite: introduction and preliminary results," in *Supercomputing'96*, 1996.

[Mil95] D.L. Mills, "Improved Algorithms for Synchronizing Computer Network Clocks," IEEE Trans. on Netwroks, 3, June 1995.

[Mol90] J.J. Molini, S.K. Maimon and P.H. Watson, "Real-time system scenarios," in *Proceedings of The 11th IEEE Real-Time Systems Symposium*, 214-225, IEEE Computer Society Press, 1990.

[Pri89] W.J. Price, “A benchmark tutorial,” *IEEE Micro*, October 1989, 28-43.

[RLLS97] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for qos management," *18th IEEE Real-Time Systems Symposium*, pages 298-307, 1997.

[Ravin98a] B. Ravindran, L.R. Welch, and B. Shirazi, “A Resource Management Middleware for Dynamic, Dependable Real-Time Systems,” Tech Report, CSE Dept, UTA, Aug. 1998.

[Ravin98b] B. Ravindran, L.R. Welch, C. Bruggeman, B. Shirazi, C. Cavanaugh, “A Resource Management Model for Dynamic, Scalable, Dependable, Real-Time Systems,” Lecture Notes in Computer Science, #1388, Springer Verlag, March 1998, pp. 931-936.

[RSYJ97] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha, "On adaptive resource allocation for complex real-time applications," In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997, 320-329.

[Ujv97] B.G. Ujvary and N.I. Kamenoff, “Implementation of the Hartstone distributed benchmark for real-time distributed systems: results and conclusions,” in *Proceedings of the 5th International Workshop on Parallel and Distributed Real-Time Systems*, 98-103, IEEE Computer Society Press, 1997.

[Wei84] R.P. Weicker, “Dhrystone: a synthetic systems programming benchmark,” *Communications of the ACM*, 27(10), 1984, 1013-1030.

[Wei89] N.H. Weiderman, “Hartstone: synthetic benchmark requirements for hard real-time applications,” CMU/SEI-89-TR-23, June 89.

[Wei90] R.P. Weicker, “An overview of common benchmarks,” *IEEE Computer*, 23(12), Dec. 1990, 65-75.

[Wei92] N.H. Weiderman and N.I. Kamenoff, “Hartstone uniprocessor benchmark: definitions and experiments for real-time systems,” *Journal of Real-Time Systems*, 4(4), Dec. 1992, 353-382.

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

[Welch97a] L.R. Welch and B. Shirazi, “DeSiDeRaTa: QoS Management Tools for Dynamic, Scalable, Dependable, Real-Time Systems,” *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, Dec. 1997, pp. 164-178.

[Welch98a] L.R. Welch, B. Ravindran, B. Shirazi, C. Bruggeman, “Specification and Modeling of Dynamic, Distributed Real-time Systems,” Proceedings of the 19th *IEEE Real-Time Systems Symposium* (RTSS’98), Dec. 1998, pp. 72-81.

[Welch98b] L.R. Welch, B. Shirazi, B. Ravindran, C. Bruggeman, “DeSiDeRaTa: QoS Management Technology for Dynamic, Scalable, Dependable, Real-time Systems,” 15th *IFAC Workshop - Distributed Computer Control Systems* (DCCS '98), Sept. 1998.

[Welch98c] L.R. Welch, et al., "Adaptive Resource Management for Scalable, Dependable Real-time Systems," in the proceedings of the Work-In-Progress Session of the Fourth IEEE *Real-Time Technology and Applications Symposium* (RTAS'98), June 1998, pp. 3-6.

[Welch98d] L.R. Welch and B. Shirazi, “A Distributed Architecture for QoS Management of Dynamic, Scalable, Dependable, Real-Time Systems”, Workshop on *Parallel and Distributed Real-Time Systems*, April 1998, available on 1998 IPPS/SPDP CD-ROM.

[Welch98e] L.R. Welch, B. Shirazi, B. Ravindran, and F. Kamangar, "Instrumentation, Modeling, and Analysis of Dynamic, Distributed Real-time Systems," Tech Rep, CSE Dept, UTA, 1998.

[ZBS97] John A. Zinky, David E. Bakken, and Richard Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, 1997.