Benchmarking Real-time Distributed Object Management Systems for Evolvable and Adaptable Command and Control Applications

Richard Freedman*, John Maurer*, Victor Wolfe, Steve Wohlever*, Michael Milligan***, Bhavani Thuraisingham***

*** The MITRE Corporation**
**** The MITRE Corporation and The University of Rhode Island**
***** U.S. Air Force Academy**

Abstract
This paper describes benchmarking for evolvable and adaptable real-time command and control systems

1. Introduction

MITRE's Evolvable Real-Time C3 initiative developed an approach that would enable current real-time systems to evolve into the systems of the future. We designed and implemented an infrastructure and data manager so that various applications could be hosted on the infrastructure. Then we completed a follow-on effort to design flexible adaptable distributed object management systems for command and control (C2) systems. Such an adaptable system would switch scheduling algorithms, policies, and protocols depending on the need and the environment. Both initiatives were carried out for the United States Air Force.

One of the key contributions of our work is the investigation of real-time features for distributed object management systems. Partly as a result of our work we are now seeing various real-time distributed object management products being developed. In selecting a real-time distributed object management systems, we need to analyze various criteria. Therefore, we need benchmarking studies for real-time distributed object management systems. Although benchmarking systems such as Hartstone and Distributed Hartstone have been developed for middleware systems, these systems are not developed specifically for distributed object-based middleware. Since much of our work is heavily based on distributed objects, we developed benchmarking systems by adapting the Hartstone system. This paper describes out effort on developing benchmarks. In section 2 we discuss Distributed Hartstone. Then in section 3, we first provide background on the original Hartstone and DHartstone designs from SEI (Software Engineering Institute) and CMU (Carnegie Mellon University). We then describe our design and modification of DHartstone to incorporate the capability to benchmark real-time middleware in Section 4. Sections 5 and 6 describe the design of the benchmarking systems. For more details of our work on benchmarking and experimental results we refer to [MAUR98] and [MAUR99]. For background information of our work we refer to [MAUR98, BENS95]).

2. DHartstone (Distributed Hartstone)

In our effort on evolving and adapting real-time command and control systems with distributed object technology, we also produced a version of a synthetic benchmark suite for real-time middleware. We obtained the source code and documentation to the Hartstone Real-time Benchmark from the Software Engineering Institute at Carnegie Mellon University [WEID89]. This benchmark was originally designed to test real-time features, such as deadlines missed under rate-monotonic scheduling, of Ada environments. Several subsequent papers have described techniques to extend the Hartstone Benchmark to distributed systems [MERC90] which is referred to as Distributed Hartstone (DHartstone). However, these distributed systems extensions were designed to determine the effects of network latency, not to benchmark middleware. We modified the design of Hartstone, while staying within the original Hartstone guidelines, to allow it to benchmark features of real-time middleware. The MITRE version of DHartstone not only provides the ability to test our adaptive middleware features, but also shows great promise in benchmarking commercial middleware for real-time applications such as AWACS.

We originally translated the Hartstone benchmarking tool from Ada to C and ported it to the LynxOS and Solaris operating systems, as described in [FREE98] to investigate the suitability of those platforms for real-time programming. We then ported the tool to our Chorus ClassiX testbed system and developed a CORBA-based distributed version (DHartstone) to test the real-time capabilities of the COOL ORB. In addition, our DHartstone code was shared with our collaborators at URI (university of Rhode Island). The URI project used our DHartstone design and code, with minor modifications, to benchmark the real-time CORBA Scheduling Service.

3. Hartstone and Dhartstone

Hartstone was originally developed by the Software Engineering Institute under contract to DoD (Ada Joint Programming Office). Ada was designed to be used for time-critical embedded applications, but it had been a matter of speculation whether Ada implementations of the time were capable of handling hard real-time applications. The Hartstone benchmark [WEID98] tests "to destruction"

system scheduling, by constructing tests with successively more difficult scheduling requirements until tasks miss deadlines.

The Hartstone benchmark consists of a number of periodic tasks at harmonic frequencies, requiring a certain fraction of system capacity. The scheduling requirements vary for successive tests in one of several ways, each designated as a particular experiment number. Experiment 1 consists of successively increasing the frequency of the highest-frequency task. Experiment 2 increases the frequency of all tasks. Experiment 3 increases the computational requirement (workload) of all tasks. Experiment 4 increases the number of tasks.

The Hartstone includes a "mini-Whetstone," a small subroutine of computationally-intensive code based originally on the Whetstone benchmark [WEID98]. The Whetstone is a *synthetic* benchmark, which means that rather than performing any useful function, it is constructed based on the instruction frequencies of a representative set of programs. The Hartstone uses a sequence of operations similar to the "official" Whetstone as a "load" for its suite of tests. It calculates the theoretical work capacity of the platform by running this subroutine a large number of times and recording the elapsed time. This, in a sense, normalizes the workload so that a faster CPU does not necessarily give a better Hartstone performance.

A test consists of a set of tasks that the main process spawns, each with an assigned period and workload. The workload is given in terms of a number of calls to the mini-Whetstone subroutine. Each task makes operating system calls to schedule itself with an assigned frequency and priority. In accordance with rate-monotonic scheduling theory, the desired frequency determines the priority, in that the higher a task's frequency, the higher its scheduling priority is. The deadline for each task is that time at which it should begin processing its next workload. Thus, upon completion of its workload, each task determines whether it has met its deadline and tallies the number of deadlines met and missed. In addition, when a deadline is missed, there is less than a full period available in which to perform the next workload. Therefore, a task will "shed load" by skipping the next one or more (if it has missed its deadline by more than a single period) workload assignment(s). Each task keeps a tally of deadlines skipped as well. The tasks keep track of, and report back to the main process at the end of a test run, their success at meeting assigned deadlines [SCHI98].

The Hartstone benchmark had been ported to the Lynx and Solaris operating systems, and translated from Ada to C, as described in [FREED98] to investigate the suitability of those platforms for real-time programming. Hartstone was also ported to the Chorus ClassiX testbed as part of the MOIE in FY98.

4. MITRE's Middleware DHartstone

In the years since the original Hartstone work was done, other investigators have extended the concept to distributed versions, providing tests with successively more difficult networking requirements. [FREE98] defines a series of requirements for a *Hartstone Distributed Benchmark* (HDB) which would integrate the processor scheduling domain with the communication scheduling domain. Those requirements were met and extended by Mercer, et al. [MERC90]. Tests were designed to increase the number, frequency, or length of messages sent among processes, until communications difficulties occurred. Thus, the effects of queuing priority, preemptability, communication latency, communication bandwidth, and packet priority could be measured.

However, all such work on Hartstone for distributed computing environments had preceded the development of ORBs. Realizing we needed some measure of effectiveness for whatever CORBA-based adaptability mechanisms we would develop, we constructed a CORBA-based distributed benchmark based on the Hartstone called DHartstone (Distributed Hartstone). Our current design is a simple variation of the classic Hartstone. The tasks spawned by the main process are the standard periodic harmonic tasks with workloads inversely related to their assigned frequencies, and the main program controls the same set of "experiments" with their respective emphasis on different aspects of the scheduler. The only difference is that instead of the tasks calling a subroutine to perform the workload, they make synchronous CORBA calls to a mini-Whetstone "server." The server spawns a thread running at the same priority as the requesting task thread. The server-side thread then executes the requested workload using the Whetstone mechanism. Once the workload is complete, the CORBA method call returns, and the client thread continues its execution.

One of the results from scheduling theory is that in a rate-monotonic system, failures will occur in a particular fashion as the system becomes overloaded. If the system is overloaded and cannot meet all deadlines, deadlines will first be missed in the lowest priority tasks. If the load increases, the failures will "work their way up" in an orderly fashion through the priority hierarchy. This is the behavior observed in the conventional Hartstone tool running on a Chorus ClassiX machine. Experiments with the DHartstone tool demonstrated that using a CORBA ORB (e.g., Chorus COOL) that is not cognizant of priorities impacts the distributed system's ability to support predictable, real-time computing. Our tests revealed that when the DHartstone test used the mini-Whetstone CORBA server for the workload, the various tasks would not fail in a predictable fashion (i.e., the highest frequency task would not be the last task to miss deadlines). This occurs even though the client-side and server-side threads are scheduled with priorities based on rate-monotonic scheduling (RMS). The fact that the distributed communication mechanism employed by COOL does not take the priorities of the requesting clients into account when scheduling the transmission of requests and replies is enough to undermine the predictable nature of RMS in DHartstone. (more details in section 6).

5. URI's Benchmarking of Real-Time CORBA Scheduling Service

URI used the basic design of MITRE's DHartstone for real-time middlware and focused on two suites of tasks based on the original Hartstone: Periodic Synchronized Harmonic

(PSH) and Periodic Synchronized Non-harmonic (PSN). Only the synchronized suites are appropriate for client/server systems like middleware. URI's Scheduling Service (see Appendix B) does not handle aperiodic tasks, so the PSH and PSN suites, which are the synchronous suites that do not include aperiodic tasks, are appropriate.

This section first specifies the parameters and describes URI's preliminary implementation of their *baseline* Fixed Priority RT CORBA Distributed Hartstone benchmark, which consists of four periodic clients and one server. Note that this report focuses on the baseline set of clients and the server, various *experiments* allow variations from the baseline of the number of clients and servers and parameters of those clients and servers. Experimental resulrs are given in [MAUR98] and [MAUR99].

We conduncted a schedulability analysis of this system using the PERTS-like schedulability analyzer under deadline-monotonic scheduling with distributed priority ceiling resource management. The output of the analysis includes *global* priority assignment for each of the clients in the system – often two priorities per client to reflect that clients have pre-period deadlines that require increased priority regions within the client. The output of the analysis also includes a global priority ceiling for each server method. In [MAUR98] we describe pseudo-code for the RT CORBA clients and servers of URI's baseline Fixed Priority RT CORBA Distributed Hartstone benchmark. This client and server code utilizes the CORBA IDL for the Scheduling Service interface that was proposed in the response to the Object Management Group (OMG) Request For Proposals (RFP) for fixed priority real-time CORBA that was submitted by Tri-Pacific/SPAWAR/ MITRE/URI (described in detail in [MAUR99]. All Clients are periodic. The current Scheduling Service prototype implementation does not yet support aperiodic clients, so they were not tested. Server tasks are executed in response to a request from a client task. Each client in the benchmark will make one method call which triggers a server task. Each client will have one pre-period deadline in addition to its period constraint

The Fixed Priority RT CORBA Distributed Hartstone benchmark contains two suites of tasks based on the original Hartstone benchmark suite: Periodic Synchronized Harmonic (PSH) and Periodic Synchronized Non-harmonic (PSN). As suggested in the original Distributed Hartstone specification, for PSH, the client period ratios are 2:4:8; and for PSN they are 3:5:7. We use client periods 2000ms, 4000ms and 8000ms for PSH; and 3000ms, 5000ms and 7000ms for PSN. Each suite has one low frequency client (8000ms in PSH and 7000ms in PSN) and one high frequency client (2000ms in PSH and 3000ms in PSN) and a variable number or medium frequency clients (4000ms in PSH and 5000ms in PSN). In the baseline there are two medium frequency clients.

The workload of both client tasks and server tasks are in even numbered *kilowhetstones*. A Whetstone is a canonical unit of floating point work [WEID89]. The number of kilowhetstones workload for both client tasks and server tasks will vary in the experiments. We use even numbers of kilowhetstones because each client workload will be split equally, with half occurring before the CORBA call and the intermediate deadline, and the other half occurring after the intermediate deadline. The baseline set of clients and servers that we implemented and executed on the prototype Scheduling Service is summarized in [MAUR98]:

URI performed a PERTS-like analysis using deadline-monotonic priority assignments and the Distributed Priority Ceiling resource management protocol. Due to the presence of an intermediate deadline, each client is split into two PERTS *tasks;* one task representing the portion of the client before the intermediate deadline, and one task for after the deadline. This split allows different priority assignments to portions of the client, which should occur in the presence of intermediate deadlines. Analysis details are given in [MAUR98] and [MAUR99].

The prototype implementation of the Scheduling Service was done on an Intel 80486 PC running the Chorus ClassiX r3.1 operating system and the Chorus COOL ORB v5 – the target real-time middleware environment of the FY98 MOIE work. The Scheduling Service is implemented as library code linked into clients and servants; there is no active daemon component to the Scheduling Service. The Scheduling Service maps the global (PERTS-generated) priorities to the priorities on the local operating system (256 local real-time priorities on ClassiX) and enforces priority ceiling semantics for dispatching execution of the server method. Each client and server implementation file is required to include a header file with definitions for the priority information used by that client or server.

6. Exploratory Experiments with Trading Object Services and DHartstone

In order to gauge the relative performance of the MITRE ClassiX/COOL Trading Object and the Real-Time Trading Object Service (for a discussion of real-time trader service, see [MAUR98]), we performed some initial experimentation. This comparison was based on the results of running versions of the DHartstone benchmarking tool that used the two types of Traders to establish client bindings to a CORBA mini-Whetstone server. That is, rather than hardcoding which CORBA server the various DHartstone client threads would use, the standard Trader or the Real-Time Trading Object Service made this decision.

As we discovered in our earlier experiments with DHartstone running on the ClassiX/COOL testbed, since the underlying CORBA infrastructure is not cognizant of the priorities the clients making CORBA requests, the DHartstone client threads' requests and replies were not scheduled based on their respective priorities. As a result, rate-monotonic behavior was not observed when either Trader is used. This is the same behavior that was seen when the client (DHartstone) was connected directly to the mini-Whetstone server (i.e., no trading object service is used). The tasks did not fail in the correct low to high priority order even when the periods and workloads were increased in an attempt to nullify the effects of the network delay and priority inversion introduced by CORBA. As a result, we were not able to demonstrate any discernable difference in

the performance of the two Traders due to the unpredictable scheduling introduced by CORBA.

As an initial effort to identify alternative mechanisms for enabling distributed, real-time computing, we modified the DHartstone tool to use ClassiX's socket mechanism, rather than the COOL ORB, to communicate between the client threads and the remote mini-Whetstone server. We configured these versions of DHartstone in a number of different ways with the goal of eventually developing an implementation similar to that used by the COOL ORB, which multiplexes all the client process's CORBA requests and replies through one communication channel e.g., a socket. See [WEID89] establishes separate request and reply sockets for each of the five client threads (see [WEID89]). This version has the most suitable design for real-time computing since using sockets per priority level greatly reduces the priority inversion in the communication mechanism. Each thread is allocated a socket to send requests to the server, and the server is allocated five unique sockets for sending replies back to the client threads. The client threads send a request to the server and then wait on their associated reply socket (i.e., clients make synchronous requests to the server). This version is designed to demonstrate that in a simple socket implementation, the tasks fail in the correct order.

With short periods (62.5msec to 1sec), the tasks do not fail in strictly rate-monotonic order (i.e., lowest to highest priority order), though it performs much more predictably than an equivalent test using CORBA. The unpredictability that does arise is most likely due to some small amount of priority inversion that still occurs in the underlying network infrastructure. Also, the server used by DHartstone is running at priority higher than any of the five client threads. This means that a high priority request that is being handled by the server may be interrupted while the server creates a new server-side thread to handle a lower priority client's request. When longer periods are used (1sec to 16secs), the tasks do fail in rate-monotonic order. We have not been able to obtain similar results using CORBA, even when we used the longer periods of execution. Details of our experiments are given in [MAUR98] and [MAUR99].

7. Summary and Directions

In this paper we have described our approach to evolving and adapting real-time distributed object management systems for command and control applications. The aspect of our work addressed in this paper is benchmarking distributed object management systems. We used the Hartstone system as our basis for developing a benchmarking system. In particular we modified the distributed hartstone system for real-time distributed object management. We discussed some experimental results for trader as well as scheduling service conducted at the MITRE Corporation and at the University of Rhode Island.

The work described in this paper is just the beginning for developing benchmarks for real-time distributed object management systems. We need to develop benchmarks to handle all of the services being specified by OMG. Subsequently we need to conduct experiments for other services also. The goal of such work is to use the benchmarks developed to determine the appropriateness of a real-time distributed object management system for various applications.

References

[BENS95] Bensley, E. et al., September 1995, *Evolvable Systems Initiative for Real-time C3: Volume 2,* Technical Report, The MITRE Corporation, Bedford, MA.

[MAUR98] Maurer, J. et al., September 1998, *Adaptable Real-time Distributed Object Management Systems: Volume 2*, Technical Report, The MITRE Corporation, Bedford, MA.

[MAUR99] Maurer, J. et al., September 1999, *Adaptable Real-time Distributed Object Management Systems: Volume 3*, Technical Report, The MITRE Corporation, Bedford, MA.

[WEID89] Weiderman, Nelson, June 1989, *Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications*, Technical Report CMU/SEI-98-TR-23, Carnegie-Mellon University, Pittsburgh, PA.

[MERC90] Mercer, C. W., Y. Ishikawa, and H Tokuda, 28 May-1 June, 1990, "Distributed Hartstone : A Distributed Real-Time Benchmark Suite," Proceedings of the 10th International Conference on Distributed Computing Systems, Paris, France

[WICH98] Wichmann, B. A., March 1988, *Validation code for the Whetstone Benchmark*, DITC 107/88, National Physical Laboratory, Teddington, Middlesex.

[FREE98] Freedman, R., R. Baldwin, P. Wallace, and T. Wheeler, June 1998, *Real Time Benchmarking of the Solaris and Lynx Operating Systems*, Technical Report 98B0000043, The MITRE Corporation, Bedford, MA.

[KAME91] Kamenoff, N. I., and N. H. Weiderman, 4-6 December, 1991, "Hartstone Distributed Benchmark: Requirements and Definitions," Proceedings of the Twelfth Real-Time Systems Symposium, San Antonio, TX., pp. 199-208.

[SCHI98] Schmidt, Doug et al., 3-5 June, 1998, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium, Denver, Colorado.

Acknowledgements: We thank MITRE's Air Force MOIE Project for supporting this work.